Supercomputing Frontiers and Innovations

2019, Vol. 6, No. 2

Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

Editorial Board

Contents

DOI: 10.14529/jsfi190201

Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community

Vladimir V. Voevodin[1], ***Alexander S. Antonov***[1], ***Dmitry A. Nikitenko***[1], ***Pavel A. Shvets***[1], ***Sergey I. Sobolev***[1], ***Igor Yu. Sidorov***[1], ***Konstantin S. Stefanov***[1], ***Vadim V. Voevodin***[1], ***Sergey A. Zhumatiy***[1]

The huge number of hardware and software components, together with a large number of parameters affecting the performance of each parallel application, makes ensuring the efficiency of a large scale supercomputer extremely difficult. In this situation, all basic parameters of the supercomputer should be constantly monitored, as well as many decisions about its functioning should be made by special software automatically. In this paper we describe the tight connection between complexity of modern large high performance computing systems and special techniques and tools required to ensure their efficiency in practice. The main subsystems of the developed complex (Octoshell, DiMMoN, Octotron, JobDigest, and an expert software system to bring fine analytics on parallel applications and the entire supercomputer to users and sysadmins) are actively operated on the large supercomputer systems at Lomonosov Moscow State University. A brief description of the architecture of Lomonosov-2 supercomputer is presented, and questions showing both a wide variety of emerging complex issues and the need for an integrated approach to solving the problem of effectively supporting large supercomputer systems are discussed.

Keywords: supercomputer, peak performance, sustained performance, efficiency, parallel computing, supercomputer center, software tools, scalability, monitoring, system level data, data analytics.

Introduction

From the very beginning of the appearance of the first computers, there were always large computing systems at Lomonosov Moscow State University. The first domestic mass-production computer Strela [1] was installed at the Computing Center of Moscow State University in 1956. Basic parameters of the machine were: 500 μs cycle time, performance of 2 thousand operations per second, 300 square meters of footprint area, power consumption of 150 kW. After Strela, there were several dozens of systems in the computer fleet of the Computing Center with various architectures, including self-developed machines based on the ternary number system.

In 1999, the first computing cluster was deployed, consisting of 12 dual-processor compute nodes connected by a fast SCI network, with a peak performance of 12 GFlops. This cluster marked the beginning of a new stage in the development of the computing resources of Moscow University, based on the active use of parallel computing technologies. These are new technologies that are more difficult to use than the conventional sequential approach, but parallel computing is a serious modern trend with enormous potential which is used in modern computers and will be used in all future computing systems.

In 2009, the first petaflops range supercomputer Lomonosov [2] produced by the T-Platforms company was installed at MSU. The supercomputer was built in several stages, and its final configuration has the following parameters: 12346 multi-core Intel processors, 2130 NVIDIA Tesla X2070/2090 graphics processors, 92 TB of RAM, QDR Infiniband as the primary interconnect, parallel data storage system, power consumption of 2.6 MW. The peak performance of the supercomputer is 1.7 PFlops, performance on the Linpack test — 901 TFlops. This is an

[1]Lomonosov Moscow State University, Moscow, Russian Federation

exceptionally large system, which requires about 1000 square meters (including the engineering infrastructure), serving hundreds of users from different organizations that solve tasks from various areas.

The huge number of hardware and software components, together with a large number of parameters affecting the performance of each application, makes ensuring the efficiency of the Lomonosov supercomputer extremely difficult. In this situation, all the basic parameters of the supercomputer should be constantly monitored, and many decisions about its functioning should be made by special means automatically. And this is not a unique feature of this particular supercomputer: an increase in the degree of parallelism and grow of complexity are objective trends of all high-performance computing systems [3]. To confirm, it is enough to analyze the list of the Top500 most powerful supercomputers in the world [4]. In practice, this fact cannot be ignored; otherwise, the efficiency of systems of this scale will be negligible. This was the reason, together with the advent of the supercomputer Lomonosov, to start research aimed at developing technologies to ensure the quality of large supercomputer systems at the Research Computing Center of Moscow State University.

The appearance of Lomonosov-2 supercomputer at MSU fully confirmed this decision. In this paper we would like to describe the tight connection between complexity of modern large high performance computing systems and special techniques and tools required to ensure their efficiency in practice. Further in the paper, Section 1 will be devoted to a brief description of the architecture of Lomonosov-2 supercomputer. In Section 2, we discuss questions showing both a wide variety of emerging complex issues and the need for an integrated approach to solving the problem of effectively supporting large supercomputer systems. Conclusion summarizes the study.

1. Lomonosov-2 Supercomputer

The first stage of Lomonosov-2 supercomputer was installed at Lomonosov Moscow State University in 2014. This system was also created by the T-Platforms company and had four stages in its development:

1. Year 2014, May: Intel Xeon E5-2680v2 10C 2.8GHz, NVIDIA K40s, 6400 cores, Infiniband FDR, peak performance 423 TFlops.
2. Year 2014, October: Intel Xeon E5-2697v3 14C 2.6GHz, NVIDIA K40s, 37120 cores, Infiniband FDR, peak performance 2.575 PFlops.
3. Year 2016, May: Intel Xeon E5-2697v3 14C 2.6GHz, NVIDIA K40s, 42688 cores, Infiniband FDR, peak performance 2.962 PFlops.
4. Year 2018, April: Intel Xeon E5-2697v3 14C 2.6GHz, NVIDIA K40s, Intel Xeon Gold 6126 12C, 2.6 GHz, NVIDIA P100, 64384 cores, Infiniband FDR, peak performance 4.946 PFlops.
5. Year 2019, June: data storage upgrade by 2.5 PBytes up to 3 Pbytes.

Since its inception in 2014, Lomonosov-2 has been included in the global Top500 ranking with the highest position #22 in November of 2014. Since spring of 2015, Lomonosov-2 steadily has been ranking #1 in the Top50 of the most powerful CIS supercomputers [5], thus confirming its leading position in the Russian supercomputer industry.

1.1. System Overview

Today Lomonosov-2 contains 1679 compute nodes in 7 racks (logically divided into three partitions "Compute", "Pascal" and "Test"), 6 management nodes, 10 service nodes, 14 distributed file system servers and 2 storage system appliances. Each compute node is an A-Class solution by the T-Platforms company. There are 7 T-Platforms A-Class racks, 6 of them are fully equipped with 256 compute nodes, and the 7th rack is partially equipped with 160 compute nodes. Infiniband and Ethernet switch systems are also installed in the A-Class system rack. All equipment in these racks excluding PSUs are liquid-cooled by hot water (up to 45 degrees Celsius inlet temperature) to provide better energy efficiency. Key parameters of the Lomonosov-2 system are presented in Tab. 1.

Table 1. Lomonosov-2 supercomputer features

Feature / Partition	Compute / Test	Pascal
Nodes	1487 / 32	160
X86 cores	20818 / 448	1920
GPUs	1487 / 32	320
Memory per node	64 GB	96 GB
GPU memory	11.56 GB	16.3 GB
GPU model	NVidia Tesla K40s	NVidia Tesla P100
CPU model	Intel Haswell-EP E5-2697v3, 2.6 GHz	Intel Xeon Gold 6126, 2.6 GHz

All compute nodes of the Compute and Test partitions have the same configuration described in Tab. 1. One rack contains up to 256 compute nodes (grouped by four on the one assembly with a single coldplate) organized into 8 pools, 2 assemblies of management node, Ethernet switch and auxiliary network Infiniband switch. Each pool contains up to 32 compute nodes (in the 8 assemblies), four 36 ports FDR Infiniband switch systems for communication network connectivity, up to 2 Ethernet switches and one FDR Infiniband switch system to provide auxiliary network connectivity. Compute nodes are connected to the switches via backplane without extra cables.

Compute nodes of the Pascal partition have the same form-factor but they are equipped slightly differently: each node of the partition has 96 GB memory, one Intel Xeon Gold 6126 processor with 12 physical cores and two NVIDIA P100 GPUs.

Mellanox dual-ports ConnectIB-based network module is installed in each compute node as well as the Gigabit Ethernet controller. There are two independent FDR Infiniband networks: communication network for MPI-like exchanges and auxiliary network for I/O operations for Lustre file system.

The communication network is used for MPI communications. Only compute nodes are connected to this network. The network is implemented using 36 ports FDR Infiniband switches which are installed in the A-Class racks. These switches are connected using the flattened butterfly topology $4 \times 8 \times 8$, which allows to extend up to 4D flattened butterfly $4 \times 8 \times 8 \times 8$. This topology was chosen for the system after different topologies simulation based on the requirement for extending the cluster up to 16K compute nodes. Each switch has 8 internal ports connected to the backplane for compute nodes connections and 28 external FDR Infiniband ports for switch-to-switch connectivity.

Management and Service network is based on the 10G/1G Ethernet protocols and used for compute nodes boot, job scheduling, monitoring and remote control. Additionally, Panasas storage system is accessible via the management network.

1.2. System Software and Programming Systems

The operating system of Lomonosov-2 is Centos-7. The only additions are Mellanox Infiniband drivers, Panasas drivers and Lustre drivers. Lomonosov-2 uses xCAT to control all boot images for all nodes and power control via IPMI.

Several OpenMPI versions are available (1.8.4, 1.10.7 and 2.1.1), as well as other MPI implementations, but only OpenMPI supports the flattened butterfly topology of the Infiniband network. Compiling can be done with GNU GCC/GFortran 4.8.5 or Intel Compiler. Intel MPI is not officially supported due to lack of support for the flattened butterfly topology.

For GPU utilization, CUDA versions 5.5, 6.5 and 8.0 are installed. Jobs control on Lomonosov-2 is secured by SLURM 15.08.1 [6] and the GLURMO custom job scheduler. System statistics are collected by collectd and nmond monitoring systems and then processed by Octotron [7] (anomaly detection). Data about actually compiled and used applications and computational packages are collected by XALT software [8].

A wide variety of preinstalled packages are available for users: abinit, espresso, lammps, namd, nwchem, vasp, cp2k, gromacs, magma, etc. Most packages are compiled with CUDA support, all of them support MPI. Intel MKL is available for users to improve performance of their applications.

Lmod [9] compatible with Environment modules was used to control environments for different versions of software.

User access to the supercomputer via ssh and sftp is possible using key-based authentication only. For users management and troubleshooting, Octoshell [10] system is actively used.

Table 2 sums up the software configuration of Lomonosov-2 supercomputer.

2. Lomonosov-2 Supercomputer and Efficiency Issues

As in any large supercomputer system with hundreds of users, there are a lot of components in Lomonosov-2 that affect the efficiency (in a broad sense) of its work, and the state of the components must be carefully controlled. Here are just some of them:

- hardware components (∼25000 units);
- software components (∼100 units);
- applications (600);
- partitions (10);
- projects (400);
- licenses (100);
- users (2500);
- organizations (300);
- queues (15);
- statuses (20);
- quotas (30);
- jobs (1000 per day)...

Table 2. Lomonosov-2 basic software components

Component	Software
Access Node OS	CentOS 7.1
Compute Node OS	CentOS 7.1
Home Filesystem	Panasas
Scratch Filesystem	Lustre 2.11
Compilers	Intel Compilers (C,C++,Fortran) 15.0; GCC Compilers (C,C++,Fortran) 4.8.5; CUDA 5.5; CUDA 6.5; CUDA 8.0
MPI	OpenMPI 1.8.4; OpenMPI 1.10.7; OpenMPI 2.1.1
Libraries	Intel MKL 2019.2
Resource Manager	Slurm 15.08.1
Job Scheduler	GLURMO
Cluster Manager	Octoshell 2
Monitoring and Analysis Tools	Collectd, nmond, Tentaviz, Octotron, XALT, DiMMoN
Packages, Libraries, Applications	Abinit, Amber, AmberTools, Athena, Charm++, CP2K, CRYSTAL-17, DL_POLY, Firefly (PC-GAMESS), Flow Vision, FMMLIB3D, Gmsh, Gromacs, Lammps, Magma, Materials Studio, Matlab, Molpro, Namd, netCFD, NWChem, Octave, OpenFOAM, Quantum Espresso, Rosetta, Schrodinger, SPILADY, Turbomole, VASP, WIEN2k, WRF...

Analyzing this list, it is necessary to take into account an important feature: there are not only many different types of components in a supercomputer, but the number of different entities within each type varies from tens of units to tens and hundreds of thousands. We already mentioned earlier a large number of components in supercomputers, and here this property becomes obvious: the numbers of entities for Lomonosov-2 are shown in brackets, and the state of each entity of each component must be controlled to ensure the supercomputer as a whole works effectively.

It may seem that some positions of this list are obvious and their processing is simple, but the guarantee of the effectiveness of a supercomputer requires not only maximum details, but also constant monitoring of changes in their state. Let us consider "licenses" and other issues related to software. For each package, library, and tool we have to keep and track all necessary details to ensure ready-to-use status for each software component:

- title and version;
- license current status;
- contacts on license;
- contacts on technical support;
- license key, license activation code;
- license expiration date;
- support termination date;
- restrictions and limitations of the license;
- license update cost, support update cost;

- path to the package, home directory;
- description of installation and fine tuning procedures, basic parameters in use;
- description of testing and checking procedures after upgrades;
- path to reference guides and users manuals;
- person responsible for installation and upgrades;
- contacts of local experts on the software;
- users, projects and organizations who are eligible to use the software.

If the license is not updated on time, or the necessary budget for software update for the next year is not allocated, or the new version of the package has not been tested by an expert in this field, then the efficiency of users, and, consequently, of the supercomputer center as a whole, decreases.

Constant control of the state of each component should be designed in such a way that at any moment it would be possible to find answers to the whole set of questions concerning the efficiency of the supercomputer. To give a feeling of a huge variety of issues that are important to control a status of a supercomputer, we give only a few examples of questions:

- What is a distribution of CPU hours consumed by different software packages for the last year? (Should we spend money for the package X next year?)
- What is average intensity of Infiniband interconnect usage for different partitions? (How large should Infiniband island be in future configurations of supercomputers?)
- How many nodes/cards/disks/cables fail every month?
- How often has Infiniband re-sent packages for the last week?
- How often does LoadAVG exceed number of cores on computing nodes?
- What is a min/max/average level of cache misses for applications of a particular user?
- What is the distribution of waiting time in queues?
- How does LoadAVG behave in my application during execution?
- Who are 5% of the most inefficient applications/users? (regarding CPU load, or LoadAVG, or cache misses, or...)
- What software packages consume 80% of supercomputer time?
- What software components being used in the supercomputer center run with efficiency less than 10%?
- What projects of the supercomputer center use Gromacs with minimal efficiency?
- What is the Top10 list of projects with the lowest CPU load?
- What is the variation in the efficiency of the supercomputer among all the projects when using the Lammps package?
- What is the Top5 list of projects of the supercomputer center by the total amount of consumed CPU hours which do not use the Infiniband interconnect network for MPI-communications?
- What is the total amount of CPU hours consumed by projects of the supercomputer center with highly intensive usage of the fast Infiniband interconnect network?
- What is the Top10 list of users by the maximum number of jobs with abnormal behavior?

There are indeed many questions that arise, since all the components mentioned above need to be considered in close relationship with each other. To guarantee complete control over the operation of supercomputers and prompt responses to all such questions, a set of tools has been developed in the RCC MSU. The main subsystems of the complex are:

- Octoshell — HPC center management system [10];

- DiMMoN — a system for deep monitoring of supercomputer parameters [11];
- Octotron — a system to ensure reliable and autonomous functioning of supercomputers [7];
- JobDigest — a visual tool to analyze the dynamic characteristics of parallel applications [12];
- an expert software system to bring fine analytics on parallel applications and the entire supercomputer to users and sysadmins [13, 14].

The subsystems of the complex are actively used on Lomonosov-2 supercomputer, providing operational data for users and administrators of the supercomputer center [15].

Conclusion

The main objective of this paper is to show the strong correlation between the high complexity of large scale HPC systems and their proper support. There are thousands of components in modern supercomputers that affect the efficiency of parallel applications, and therefore they all require constant deep monitoring. The increasing complexity of computer architecture and the growth of the degree of parallelism are characteristic features that are typical for all, without exception, modern large supercomputer systems. This fact must necessarily be taken into account in any supercomputer center, otherwise its productivity will be in doubt. A set of advanced software tools aimed at solving this problem was developed at MSU Research Computing Center, and the first experience of its use on Lomonosov-2 supercomputer showed both the correctness of the proposed approach and the need to continue and expand work in this direction in the future.

Acknowledgements

The results were obtained with the financial support of the Russian Foundation for Basic Research (grant No. 18-29-03230). The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

References

1. Strela (in Russian). `http://www.computer-museum.ru/histussr/strela0.htm`, accessed: 2019-06-20

2. Sadovnichy, V., Tikhonravov, A., Voevodin, Vl., Opanasenko, V.: "Lomonosov": Supercomputing at Moscow State University. In: Contemporary High Performance Computing: From Petascale toward Exascale (Chapman & Hall/CRC Computational Science), pp. 283–307. Boca Raton, USA, CRC Press (2013)

3. Dongarra, J., Beckman, P. et al.: The International Exascale Software Roadmap. International Journal of High Performance Computer Applications 25(1), 3–60 (2011), DOI: 10.1177/1094342010391989

4. TOP500 Supercomputer Sites. `https://www.top500.org/`, accessed: 2019-06-20

5. Top50 supercomputers of Russia (in Russian). http://top50.supercomputers.ru/, accessed: 2019-06-20

6. Slurm workload manager. http://slurm.schedmd.com/slurm.html, accessed: 2019-06-20

7. Antonov, A., Nikitenko, D., Shvets, P., Sobolev, S., Stefanov, K., Voevodin, Vad., Voevodin, Vl., Zhumatiy, S.: An approach for ensuring reliable functioning of a supercomputer based on a formal model. In: Parallel Processing and Applied Mathematics. 11th International Conference, PPAM 2015, Krakow, Poland, September 6–9, 2015. Revised Selected Papers, Part I Lecture Notes in Computer Science, vol. 9573, pp. 12–22. Springer International Publishing (2016), DOI: 10.1007/978-3-319-32149-3_2

8. Agrawal, K., Fahey, M.R., McLay, R., James, D.: User environment tracking and problem detection with XALT. In: Proceedings of the First International Workshop on HPC User Support Tools, 21–21 Nov. 2014, New Orleans, LA, USA. pp. 32–40. IEEE Press (2014), DOI: 10.1109/HUST.2014.6

9. McLay, R.: Lmod: Environmental Modules System. http://www.tacc.utexas.edu/tacc-projects/lmod, accessed: 2019-06-20

10. Nikitenko, D., Voevodin, Vl., Zhumatiy, S.: Resolving frontier problems of mastering large-scale supercomputer complexes. In: Proceedings of the ACM International Conference on Computing Frontiers (CF'16), May 16–19, 2016, Como, Italy. pp. 349–352. ACM New York, NY, USA (2016), DOI: 10.1145/2903150.2903481

11. Stefanov, K., Voevodin, Vad., Zhumatiy, S., Voevodin, Vl.: Dynamically Reconfigurable Distributed Modular Monitoring System for Supercomputers (DiMMon). In: 4th International Young Scientist Conference on Computational Science. Procedia Computer Science, vol. 66, pp. 625–634. Elsevier B.V Netherlands (2015), DOI: 10.1016/j.procs.2015.11.071

12. Nikitenko, D., Antonov, A., Shvets, P., Sobolev, S., Stefanov, K., Voevodin, Vad., Voevodin, Vl., Zhumatiy, S.: Jobdigest — detailed system monitoring-based supercomputer application behavior analysis. In: Third Russian Supercomputing Days, RuSCDays 2017, Moscow, Russia, September 25–26, 2017, Revised Selected Papers. Communications in Computer and Information Science (CCIS), vol. 793, pp. 516–529. Springer Cham (2017), DOI: 10.1007/978-3-319-71255-0_42

13. Nikitenko, D., Shvets, P., Voevodin, Vad., Zhumatiy, S.: Role-dependent resource utilization analysis for large HPC centers. In: Parallel Computational Technologies. Communications in Computer and Information Science (CCIS), April 2–6, 2018, Rostov-on-Don, Russia. vol. 910, pp. 47–61. Springer (2018), DOI: 10.1007/978-3-319-99673-8_4

14. Shaykhislamov, D., Voevodin, Vad.: An approach for detecting abnormal parallel applications based on time series analysis methods. In: Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science, September 10–13, 2017, Lublin, Poland. vol. 10777, pp. 359–369. Springer International Publishing (2018), DOI: 10.1007/978-3-319-78024-5_32

15. Nikitenko, D., Voevodin, Vad., Zhumatiy, S.: Deep analysis of job state statistics on Lomonosov-2 supercomputer. Supercomputing Frontiers and Innovations, 5(2), 4–10 (2018), DOI: 10.14529/jsfi180201

DOI: 10.14529/jsfi190202

HPC Processors Benchmarking Assessment for Global System Science Applications

Damian Kaliszan[1]***, Norbert Meyer***[1]***, Sebastian Petruczynik***[1]***,***
Michael Gienger[2]***, Sergiy Gogolenko***[2]

The work undertaken in this paper was done in the Centre of Excellence for Global Systems Science (CoeGSS) – an interdisciplinary project funded by the European Commission. CoeGSS project provides a computer-aided decision support in the face of global challenges (e.g. development of energy, water and food supply systems, urbanisation processes and growth of the cities, pandemic control, etc.) and tries to bring together HPC and global systems science. This paper presents a proposition of GSS benchmark which evaluates HPC architectures with respect to GSS applications and seeks for the best HPC system for typical GSS software environments. The outcome of the analysis is defining a benchmark which represents the average GSS environment and its challenges in a good way: spread of smoking habits and development of tobacco industry, development of green cars market and global urbanisation processes. Results of the tests that have been run on a number of recently appeared HPC platforms allow comparing processors' architectures with respect to different applications using execution times, TDPs[3] and TCOs[4] as the basic metrics for ranking HPC architectures. Finally, we believe that our analysis of the results conveys a valuable information to the broadened GSS audience which might help to determine the hardware demands for their specific applications, as well as to the HPC community which requires a mature benchmark set reflecting requirements and traits of the GSS applications. Our work can be considered as a step into direction of development of such mature benchmark.

Keywords: Global Systems Science, HPC benchmarks, parallel applications, e-Infrastructure evaluation.

Introduction

Global Systems Science (GSS) is a branch of science which uses specific knowledge and techniques to evaluate the impact of policies and people's relation on various global phenomena such as climate change, financial crises, pandemic spread, growth of the cities, human migration, etc. This document addresses the question of "which HPC architectures among the recently introduced are best to run GSS applications most effectively?". Such aliases as "the best" or "most effectively" may obviously have different meanings for different people. While some people might consider it to be the fastest execution time, others might be interested in the price-performance ratio calculated as the price of a processor multiplied by total execution time (for given architecture) or the least carbon footprint left, calculated as a product of TDP and total execution time. For the purpose of this study, the authors acquired cutting-edge processors from four major vendors – Intel, AMD, HiSilicon, and IBM. In particular, we benchmarked GSS applications on Intel®Xeon®Gold 6140 [31] 2-node cluster, AMD Epyc^{TM} 7551 single node, ARM Hi1616 2-node cluster, IBM Power8+ [28] single node, and – as a reference testbed – Eagle cluster located at Poznan Supercomputing and Networking Center (Poland) equipped with Intel®Xeon®Haswell E5-2697 v3 processors. The set of tested applications (called a *GSS benchmark* here) covers many research areas from the entire GSS field. The rest of the paper is organised as follows. Section 1 describes benchmark and the experimental setup. More specifically, we introduce applications chosen

[1]Poznan Supercomputing and Networking Center, Poznań, Poland
[2]High-Performance Computing Center Stuttgart, Stuttgart, Germany
[3]Thermal Design Power
[4]Total Cost of Ownership

for the benchmark and substantiate this choice, present a technical overview of the testbeds where the tests were launched, and discuss the approach selected for measuring the performance metrics. In Section 2, we shortly overview the main benchmarking results. In particular, we emphasise applicability of those results for the hardware/software co-design. Section 3 contains statements for the most relevant conclusions. Finally, we end our paper with a formulation of the directions for future research.

1. Benchmark and Experimental Setup

1.1. Representative GSS Applications Selected for the Benchmark

Functionally, the benchmarked applications can be categorized into two groups: HPC- compliant social simulation software and large-scale CFD (Computational Fluid Dynamics) applications. From the perspective of programming languages, GSS benchmark covered applications are written in C++ and Python, which are the most popular programming languages among GSS experts who use HPC. This subsection contains a short description of tested applications – ABM4Py/GG, Pandora/GG, IPF, OpenSWPC, CCTM, CM1, HWRF – and explains why these particular applications were selected for the benchmark. We refer the interested reader for further technical details to the reports [23, 24].

1.1.1. HPC-compliant Social Simulation Software

Since GSS deals with evaluating impact of different policies on the society, social simulations constitute a relevant part for the majority of workflows in large-scale GSS applications. Typical social simulation component consists of pre-processing, simulation, and post-processing steps. Agent-based modelling and simulation (ABMS) is the most widely accepted tool for the simulation step. Pre-processing step takes care of collecting and wrangling real-word data, as well as synthesising inputs for ABMS. As long as fine-grained data about society are rarely available for public use, large-scale agent-based models usually require synthetic input data – synthetic populations and synthetic social networks – prepared on the base of partial and marginal information. Post-processing includes visualisation, data analytics, model verification, validation and uncertainty quantification.

In our benchmark, the group of social simulation software covers applications for pre-processing and simulation of agent-based models. In the benchmark definition, we intentionally skipped post-processing step as it frequently heavily depends on the concrete GSS problems in hand, so it is hard to determine typical use-cases and computational kernels for post-processing. We selected two ABMS frameworks – ABM4Py is implemented in Python and follows graph-based ABMS methodology particularly suitable for large-scale social simulation, while Pandora is written in C++ and uses regular meshes to simulate the environment. The pre-processing step is represented by HPC-compliant implementation of the IPF method for generating synthetic populations. The text below shortly describes each application and the specific inputs used into benchmark.

IPF (iterative proportional fitting) is a "technique that can be used to adjust a distribution reported in one data set by totals reported in others. IPF is used to revise tables of data where the information is incomplete, inaccurate, outdated, or a sample" [6]. This procedure reconstructs a contingency matrix based on the known marginals in an unbiased way. Nowadays, IPF and

its derivatives (e.g., IPU) constitute the computational core of the most popular techniques for generating synthetic data which serve as an input for agent-based social simulations [18].

Despite its popularity, we are not aware of any HPC-compliant open-source implementation of IPF. In order to overcome this obstacle, we coded a simple IPF process using linear algebra kernels from ScaLAPACK. This simple implementation was a baseline for our IPF benchmark.

Note: The above-mentioned IPF codes are not published with open access on the Internet yet.

ABM4Py/GG (ABMS for Python) is a distributed agent-based modelling and simulation framework for fast prototyping agent based components of GSS models [17]. Agent-based models implemented in ABM4Py follow the graph-based representation [17, 29]. Namely, agents and loci of interactions are interconnected into a complicated dynamic network, and agent-based simulation reflects possible temporal evolution of this network. This network is further split into subgraphs with graph partitioning software and distributed between MPI processes.

In order to benchmark this framework, we implemented the green growth agent-based model, first proposed in [7], within ABM4Py framework. The green growth model is an innovation-diffusion model for electric cars with a global scope and a fine-scale spatial data resolution. This model allows measuring the most relevant performance metrics for ABM4Py, including elapsed time, I/O, waiting time, and synchronisation time. In order to enable comparison of the ABM4Py framework with Pandora frameworks (see below), our toy implementation uses 2D mesh as a topology of the environment. More specifically, we test against two cases – the one where layer-shape size is set to 64x64, and the second with a size of 128x128. The project's repository is available at [19].

Pandora/GG application serves for benchmarking of Pandora agent-based modelling and simulation framework. In contrast to ABM4Py, Pandora only supports raster inputs, which restricts this framework to agent-based models with 2D mesh as a topology of the environment. Pandora parallelises the simulation process via splitting of rasters on even pieces and distributing them between MPI processes. On the other hand, Pandora is implemented in C++, which allows reaching higher performance compared to Python-based ABM4Py framework.

Similarly to the ABM4Py use-case, our Pandora/GG application implements the green growth agent-based model from [7]. The project's repository is available at [12].

1.1.2. Large-scale CFD Applications

A wide variaty of GSS applications – from evacuation planning [21] to air quality control [15] – rely on coupling of ABMS with CFD. The group of benchmarked CFD applications includes large scale tools that simulate GSS-related scenarios like natural disasters (hurricanes, earthquakes), spread of air pollution, and weather forecast. We selected 4 exemplar open-source codes for large-scale CFD simulations:

- OpenSWPC – an integrated parallel simulation code for modelling seismic wave propagation in 3D heterogeneous viscoelastic media which is applicable for evacuation planning in case of earthquakes;
- CMAQ – a community multiscale air quality modelling system, which was successfully used for conducting large scale air quality simulations and policy making for air pollution control [15];

- CM1 – a model for studying processes in the Earth's atmosphere, which can be used for weather forecast in different GSS applications where behaviour of agents in ABMS component of GSS model depends on weather (e.g., predicting refugee destinations [30]);
- HRWF – parallel implementation of the hurricane weather research and forecasting which is suitable for evacuation planning in case of hurricanes.

OpenSWPC (Open-source Seismic Wave Propagation Code) is an open-source software from large-scale simulation of seismic waves propagation (2D or 3D) by solving motion equations using the finite difference method (FDM) [8]. OpenSWPC is widely used in seismology. It ports easily and delivers good performance on different distributed systems varying from small PC clusters to large-scale supercomputers. The project's repository is available at [10].

CMAQ/CCTM (Community Air Multiscale Quality Modelling System) is an active open-source project of the U.S. EPA (*Environmental Protection Agency*) that delivers a suite of programs for conducting the air quality model simulations. CCTM (*The CMAQ Chemistry-Transport Mode*) is a parallel implementation of the advanced chemical transport model in CMAQ which is often used in computer-aided policy making for improving air quality [15]. It is the only CMAQ program that can be run in parallel.

CCTM runs require large input datasets with a complicated file structure. In our study, we used the official single day simulation benchmark dataset distributed by EPA [5]. Both the project description and the application are available at [1].

CM1 is a three-dimensional, time-dependent, non-hydrostatic numerical model. CM1 is designed primarily for idealized research, particularly for deep precipitating convection and for studies of relatively small-scale processes in the Earth's atmosphere, such as thunderstorms [11]. Both the project description and the application are available at [9].

WRF (Weather Research and Forecasting model) is an example of a well-scalable application, which motivated us to add it to the set of tests in order to increase the variety of requirements of the GSS benchmark. The Hurricane Weather Research and Forecasting (HWRF) model is a specialised version of the WRF model [16]. It is used to forecast the track and intensity of tropical cyclones. Both the project description and the application are available at [2].

This document does not go further into theory as it is beyond its main subject.

1.2. Configuration of Testbeds Used for the Benchmark

We executed our benchmark on four testbeds using cutting-edge processors recently introduced by four major processor vendors: Xeon®Gold 6140 from Intel [31], AMD Epyc™ 7551 from AMD [3, 25], ARM Hi1616 from HiSilicon [4, 20], and Power8 from IBM [13, 14]. While Tab. 1 summarises relevant technical characteristics of our testbeds, the following paragraphs describe important architectural improvements introduced into the processors.

Intel® Xeon® Gold 6140 (SkyLake SP). The new core for Skylake-X, technically called the Skylake-SP core architecture, delivers new improvements compared to the previous Broadwell-E platform. One of those "upgrades" has been targeted at a better SIMD performance: clustering multiple data entries into a single element and performing the same operation to each of them at

Table 1. Testbed characteristics

	Intel® Xeon® Gold 6140	AMD Epyc™ 7551	ARM Hi1616	Power8+ S822LC
No. of nodes	2	1	2	1
Cores per node	36	64	64	20
CPU Frequency [GHz]	2.3	2	2.4	2.92
L1 (data) cache	1.125 MB	2 MB	2 MB	1.25 MB
L2 cache	36 MB	32 MB	16 MB	10 MB
L3 cache	49.5 MB	128 MB	64 MB	160 MB
RAM type (channels)	DDR4 (6)	DDR4 (8)	DDR4 (4)	DDR3 (4)
RAM frequency [MHz]	2666	2400	2400	1333
RAM capacity [GB]	192	512	128	512
I/O and disks type	SSD	SSD	SSD	SSD
Network	10Gb Ethernet	–*	10Gb Ethernet	–*
TDP [W]	140	180	70	190
Processor price [USD]	2450	3743	300	1500
OS version	Ubuntu 16.04.03 LTS	CentOS 6.9	EulerOS release 2.0 (SP2), Ubuntu 16.04.3 LTS	Ubuntu 16.04.2 LTS
Total bandwidth estimates (per node) [GBps]				
L1 (data) cache	15897	6144	19660	2803
L2 cache	5299	8192	19660	2803
L3 cache	5299	8192	19660	3738
Total memory	119.21	158.95	71.53	230
SMP interconnect	62.4	37.92	48	38.4
I/O (maximum theoretical, simplex**)	96	128	92	64

* These testbeds have only one node, therefore, network is not used in this case;

** All testbeds use PCIe interconnect, thus, for total I/O bandwidth at full-duplex simply multiply by 2.

once in one go. This has evolved in many forms, from SSE and SSE2 through AVX and AVX2 and now into AVX-512-F. Other important changes available in Intel® Xeon® Gold are presented separately in [31].

AMD Epyc™7551. This processor is based on the Zen microarchitecture and is manufactured on a 14 nm process. This microarchitecture was designed from the ground up with data centres in mind, for optimal balance and power. The new core design can process four x86 assembler instructions per cycle and introduces Simultaneous Multithreading (SMT). Zen microarchitecture also introduces a considerable number of improvements and design changes over Bulldozer including wider instruction set, larger cache system, 2x higher bandwidth, better branch predictions, etc [3, 25].

ARM Hi1616. The HiSilicon Hi1616 V100 products are based on ARM Cortex-A72 cores. These are high-performance, low-power processors based on the ARMv8-A architectural platform. Hi1616 features several major microarchitectural improvements in memory performance, as well as in integer and floating point arithmetics that build on top of the current generation of ARMv8-A cores [4, 20].

Power8+ S822LC. Being designed for "accelerated workloads in high-performance computing (HPC), high-performance data analytics (HPDA), enterprise data centers, and accelerated cloud deployments" [13, 14], IBM 8335 Power System S822LC for High Performance Computing server Model GTB (8335-GTB) perfectly suits for all kinds of GSS applications. S822LC brings together two POWER8 CPUs with four NVIDIA Tesla P100 GPUs through novel NVLink Technology.

1.3. Measuring and Reporting the Performance Technique

In the process of preparing and launching the benchmark, we faced a number of technical difficulties, which significantly influenced the approach we have chosen to measure and report performance. This subsection highlights our approach to tackling those difficulties.

Since the analysed testbeds represent brand-new architectures, we encountered a limited number of tools for measuring metrics of interest which were strait available for all testbeds. In particular, many popular performance measuring tools – VampirTrace, Scalasca, etc – were not ported to the ARM Hi1616 by the time of performing benchmark. Moreover, tested applications are written in different languages – C/C++, Fortran, and Python – which reduces the range of performance measuring tools suitable for all applications at once. On the other hand, our study is focused on overall performance of the code and does not require instrumentation to measure and analyse performance. Thus, we decided to omit specialised benchmarking libraries (e.g., LibSciBench [22]) and performance analysis tools (e.g., VampirTrace) in favour of standard Linux toolset available out-of-the-box. The metrics of interest were measured by means of `/usr/bin/time` Linux utility. This utility allows measuring the following metrics for the application as a whole and for each separate MPI process: total elapsed real time, the number of filesystem inputs and outputs, maximum resident set size of the process, average total (data+stack+text) memory use of the process, etc. We used measurements of the above-mentioned metrics and the data from Tab. 1 to compute dependent metrics (like TDP and TCO) and to build all charts and plots in this paper. In order to keep conditions of the experiments even, the system caches were flushed by calling `sync; echo 3 > /proc/sys/vm/drop_caches` before each experiment. All C/C++ and Fortran codes were compiled with gcc version 5.4.0 using -O3. Python 3.5.2 was used as a Python interpreter for the ABM4Py application.

In addition, we experienced issues with different orders of magnitude in elapsed times of the applications. Even though we adjusted the input files to keep the elapsed time scales closer for all application, we still had significant differences in total elapsed times between social simulation and CFD software: each individual benchmark for social simulation codes consumed less than 3 hours on any testbed, while benchmarks for some CFD application required more than 30 hours. The latter fact prevented us from performing many repetitions of time-consuming CFD application runs. More specifically, in order to reduce the number of repetitions, we started with 2 runs and repeated the experiment until the ratio of the sample standard error to the sample mean of the elapsed time was less than 5% in each test configuration. Since the number of measurements generated with this approach, it is sometimes insufficient to construct meaningful confidence interval as suggested in the performance reporting guidelines [22], Fig. 1 and 2 report sample means of the measurements without confidence interval.

Last but not least, our testbeds were limited by 1–2 nodes, which is not enough to conduct full scalability experiments with some of the selected applications. In order to overcome this obstacle, we used the Eagle cluster equipped with 50 nodes containing 2, 14-core Intel Haswell E5-2697 v3 CPUs each, as a reference testbed, where we performed tests up to reaching the scalability bound. This allowed us to get an impression on the scalability of the applications reflected in Tab. 2.

2. Benchmarking Results

Figures 1 and 2 summarise major benchmarking results for social science and CFD applications respectively. For each application, we include two plots: the first one – scalability plot –

illustrates the change in the total elapsed time with the grows of the number of MPI processes for all testbeds, whereas the second one – the metrics plot – presents all metrics measured for Intel Xeon Gold 6140 with different number of MPI processes. Intel Xeon Gold 6140 has been chosen to report details on measured metrics since it demonstrated the best performance compared to other testbeds (see Section 3). Besides the plots for Intel Xeon Gold 6140, reports [24] and [23] contain similar plots and supplementary information for the remaining testbeds. Both Fig. 1 and 2 share identical legends. Note that scalability plot for HWRF application (Fig. 2g) lacks information about ARM Hi1616 and IBM Power8+. We failed to port HWRF on those architectures.

2.1. HPC-compliant Social Simulation Software

Our benchmarks demonstrate high performance of IPF on different architectures. The application scales to a number of available cores on all testbeds in out study (see Fig. 1a). On the reference testbed, we observe the scalability bound for more than 1400 cores. Neither RAM, nor I/O of modern architectures are the limiting factors for IPF performance (see Fig. 1b). The reason of such good results is in a heavy use of highly optimised ScaLAPACK kernels in the IPF implementation. In contrast to other applications, in case of IPF, the least elapsed time is observed for AMD Epyc^{TM} 7551 testbed (see Fig. 1a).

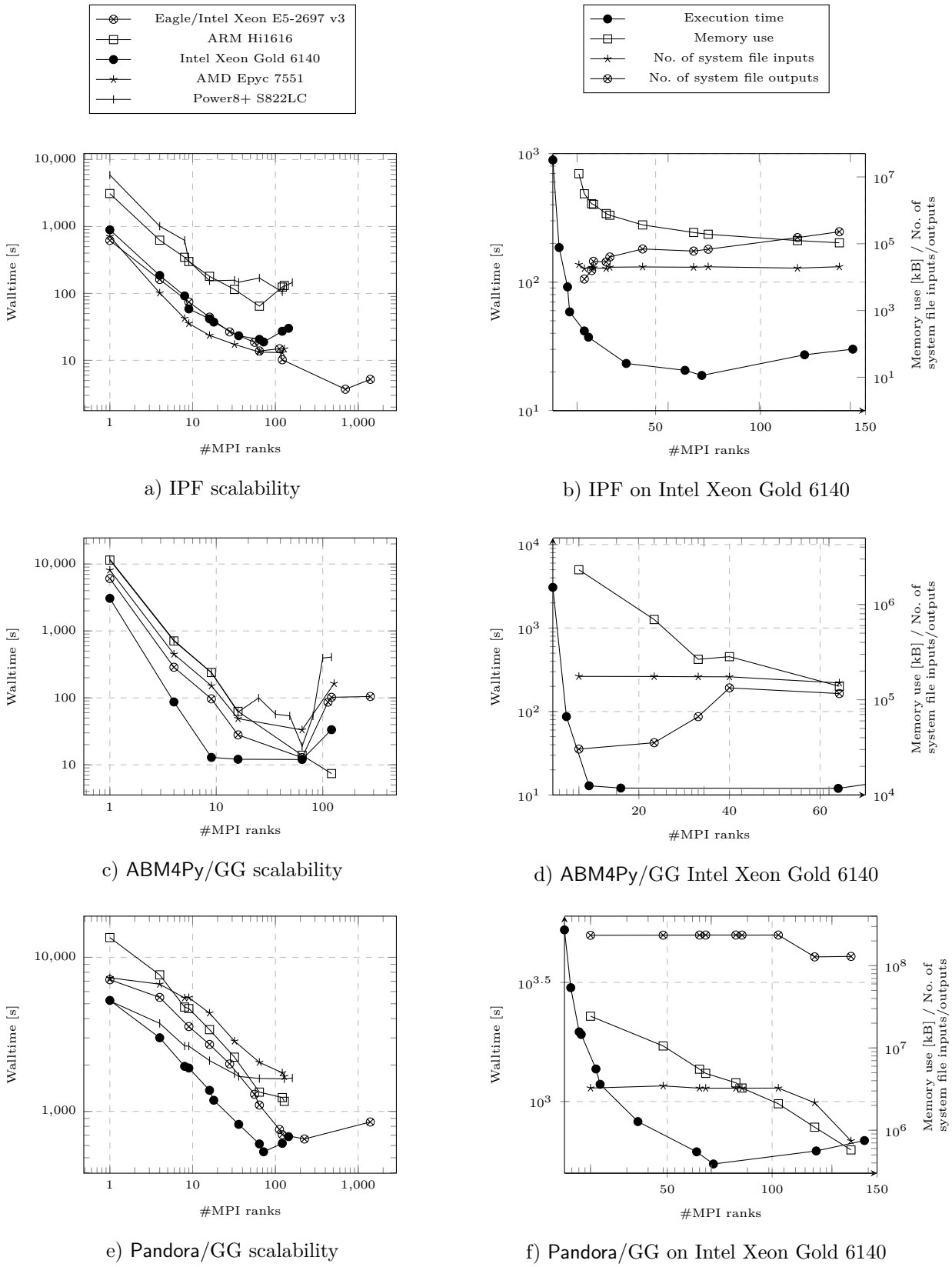
Along with IPF, we benchmarked a green growth agent-based model (ABM) of diffusion implemented in ABM4Py and Pandora frameworks.

In both cases, despite a strong difference in parallelization strategies, we observe the same pattern: ABMS applications produce a big amount of output which has a strong negative impact on application performance (see Fig. 1d and 1f). As a consequence, according to our green growth ABM, being I/O bound, current ABMS frameworks for HPC have moderate requirements to CPU performance. Nevertheless, we must emphasise that the results can look differently for more complex models with sophisticated agent activities and for simpler models which can be reduced to iterative applying of sparse matrix-vector or matrix-matrix operations (e.g., random surfer model and PageRank). Thus, our benchmarks for ABMS frameworks are not very illuminative and must be extended with more sophisticated and more simple models to provide more evidences and draw stronger conclusions. But discussion of the new representative ABMS models for benchmarking goes beyond the scope of this text.

2.2. Large-scale CFD Applications

Our measurements demonstrate that CFD applications are in general CPU-bound (see Fig. 2). Nevertheless, we observed that at some architectures, memory is also a bottleneck for some specific choices of the number of MPI processes. In particular, we noticed that OpenSWPC is memory-bound for a small number of MPI processes and CPU is bound for a large number of utilised cored, as the memory consumption monotonically decreases with the number of MPI processes in this application (see Fig. 2b). All benchmarked applications except for HWRF demonstrate the same monotonic decrease in memory consumption (see Fig. 2). At the same time, system files' outputs make a solid contribution to the total elapsed time for such applications as CCTM and CM1 (see Fig. 2d and 2f), which, in turn, imposes additional performance constraints on architectures with low I/O bandwidth.

Note: Both figures 1 and 2 share identical legends



a) IPF scalability

b) IPF on Intel Xeon Gold 6140

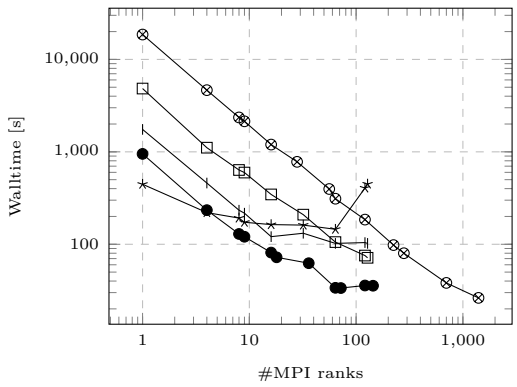
c) ABM4Py/GG scalability

d) ABM4Py/GG Intel Xeon Gold 6140

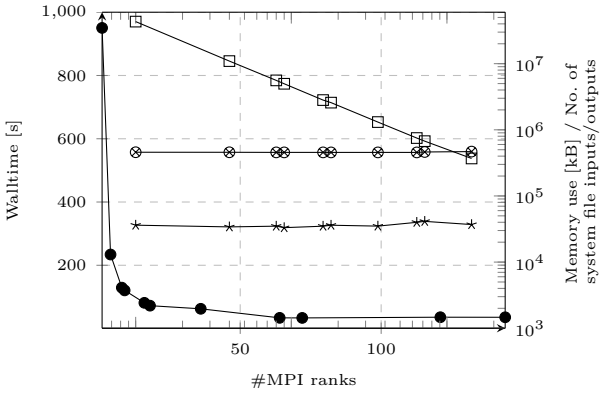
e) Pandora/GG scalability

f) Pandora/GG on Intel Xeon Gold 6140

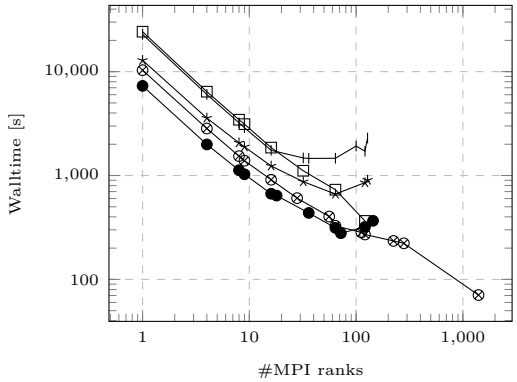
Figure 1. Results for social simulation applications



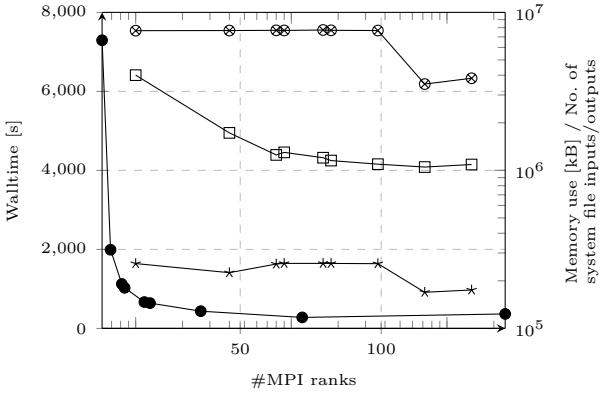
a) OpenSWPC scalability



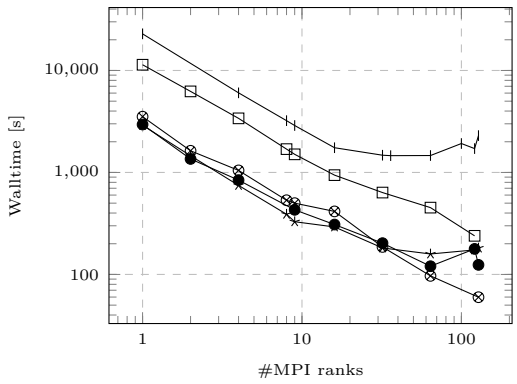
b) OpenSWPC on Intel Xeon Gold 6140



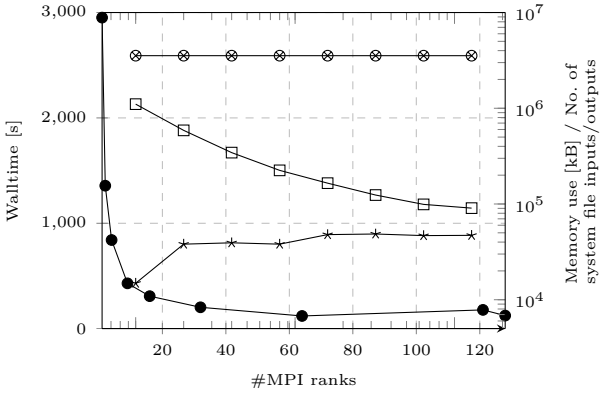
c) CMAQ/CCTM scalability



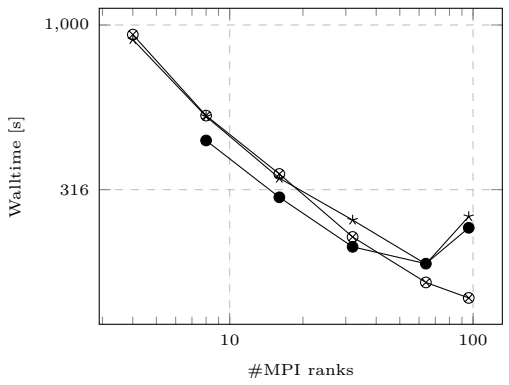
d) CMAQ/CCTM on Intel Xeon Gold 6140



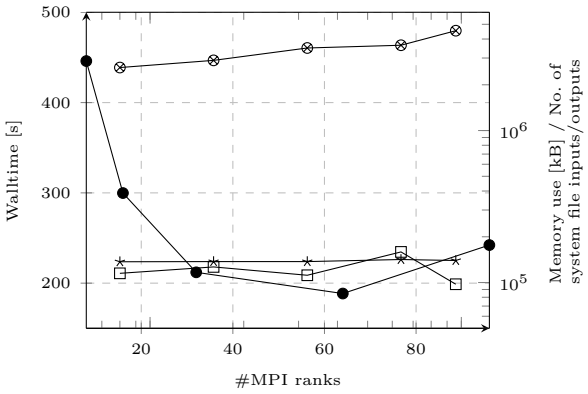
e) CM1 scalability



f) CM1 on Intel Xeon Gold 6140



g) HWRF scalability



h) HWRF on Intel® Xeon® Gold 6140

Figure 2. Results for CFD applications

Intel® Xeon® Gold 6140 provides the least elapsed time for all CFD applications from our benchmark with one minor exception (see Fig. 2). AMD Epyc™ 7551 beats Intel® Xeon® Gold 6140 in OpenSWPC for the small number of cores (see Fig. 2a). Nevertheless, performance of AMD Epyc™ 7551 degrades quickly compared to Intel® Xeon® Gold 6140 if the number of utilised cores grows.

2.3. Implications for the Hardware/Software Co-design

Table 2 shortly summarises information about scalability of the benchmarked social simulation software and about hardware bottlenecks revealed in the previous subsection. In this table, rows from group "Bottlenecks" reflect which of the following – CPU, memory consumption, system files' inputs or system files output – appeared to be limiting factors for performance of the benchmarked applications, whereas the "Scalability" row presents the maximum number of utilised CPU cores where we observed a decrease in the total elapsed time on the reference testbed – the Eagle cluster.

Table 2. Bottlenecks in the hardware and scalability of the benchmarked applications

		Social simulation software				CFD software			
		IPF	Pandora		ABM4Py	HWRF	OpenSWPC	CMAQ CCTM	CM1
			Europe	World	128x128				
Bottlenecks	CPU	✓				✓	✓	✓	✓
	RAM						✓		
	Inputs					✓			
	Outputs		✓	✓	✓			✓	✓
Scalability*		≈700	≈128	≈700	≈64	≥128	≥128	≥128	≥128

* maximum number of utilised cores of Xeon E5-2697 v3 cluster that leads to reduction of the total elapsed time.

As Tab. 2 illustrates, most of the distributed GSS applications are memory-bound. Even large-scale CFD codes can be bound by I/O and RAM under special circumstances. It allows us to conclude that the fast memory is an essential requirement to HPC clusters for GSS applications whereas high CPU clock frequency plays a less important role. Moreover, since many state-of-the-art GSS applications deal with large input and output files, we believe that GSS software developers should invest more time into design of file-avoiding applications. Our scalability tests show that hyper threading provides little performance improvements for most of the GSS applications. Therefore, it makes little sense to invest money in expensive massively multithreaded chips (like Power8) for GSS users. We also recommend avoiding clusters with GPU accelerated nodes since only a few popular GSS applications benefit from GPUs. In particular, among widely used general-purpose ABMS frameworks and problem-specific ABMS codes for HPCs, only the FLAME-GPU (Flexible Larg-scale Agent Modelling Environment) framework utilises GPUs [26, 27]. Seldom use of GPUs is also partially related to the fact that most social science applications are memory-bound. Being more specific, among the architectures used in benchmarking, we recommend to build clusters upon ARM Hi1616 in case that energy efficiency is a crucial requirement, or upon Intel® Xeon® Gold 6140 in case that performance is a first priority while relatively high operating expense and capital expenditure are not an issue.

According to our benchmarks, the scalability of GSS applications is rather diverse. All applications from the social simulation software stack demonstrate poor scalability with one notable exception – the IPF implementation. Moreover, even though our benchmarks do not demonstrate this explicitly, it is also known that social simulation software scales are worse than the large-scale

CFD codes. On the other hand, due to stochastic nature of ABMs, a typical social simulation workflow assumes many simultaneous simulation runs, whereas the fitting step in reconstruction of a synthetic population should normally be performed only once for a given dataset. Therefore, the optimal number of nodes for the state-of-the-art should be defined by scalability of the synthetic population and CFD codes (if the latter ones are of interest for the target GSS audience). We can always bypass the gap in scalability of the synthetic population and ABMS codes and reach full utilisation of clusters by making several simultaneous simulation runs (and treating simulation results in a file-avoiding way).

Unfortunately, our results do not allow for drawing solid conclusions about node interconnections since the benchmarks had been done on the testbeds with only one or two nodes.

3. Discussion

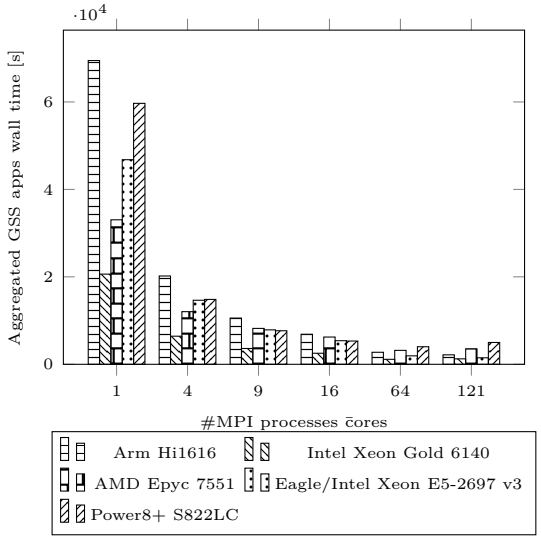
The findings of this work allowed the authors to formulate the following conclusions:

- Among all tested applications, IPF is the least I/O demanding. For the reference architecture it shows, the scalability is up to 700 cores. On the other hand, other selected processors scale in the range of a number of physical cores, so we expect that using them in a multinode configuration will result in a behaviour similar to the reference testbed.
- Green Growth-pilot applications are dominated by I/O operations (mostly output) where a large HDF5 file is created and to which all processes save data;
- Results obtained for ABMS4py – another social simulation application-prove that it should be subject to major improvements. For instance, the best execution time on Intel® Xeon® Gold 6140 is observed for only 9 MPI processes, whereas the testbed includes 72 physical cores (2-node configuration with 2 chips each, 18 cores per one chip)
- OpenSWPC benchmark for given input configuration reports good scalability. It is especially illuminative in case of the reference testbed where the execution time decreases along the number of cores used until the maximum number of 1400 is used. Other processors show similar behaviour. Using hyper threads (where possible) does not provide any further time improvements.
- CCTM is I/O dominated (especially output) application and, thus, the impact on the execution times is high
- CM1 results indicate a relationship between the problem size (weather map grid) and the processors mesh, as well as the dependence between the number of threads and nesting of cores in different levels of cache.
- HWRF demonstrates very similar results to CM1 in terms of scalability on processors equipped with the implementation of simultaneous multithreading
- Best execution time gets usually achieved (when considering single nodes) for the number of processes equal to the number of cores: 64 for 2-node ARM Hi1616 and 1-node AMD Epyc^{TM}, 72 for 2-node Intel® Xeon® Gold 6140, 20 for 2-processor 1-node Power8+;
- In most cases, hyper threading does not bring any performance improvements.

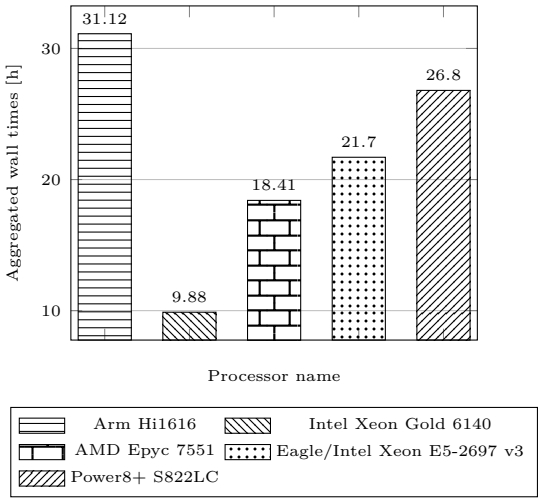
For the final comparative analysis, two additional metrics have been introduced:

- *Energy efficiency* calculated as a product of walltimes and TDP products which scales and binds the achieved timing results by processors by the theoretical heat generated during the tests and/or the energy consumed by processors.

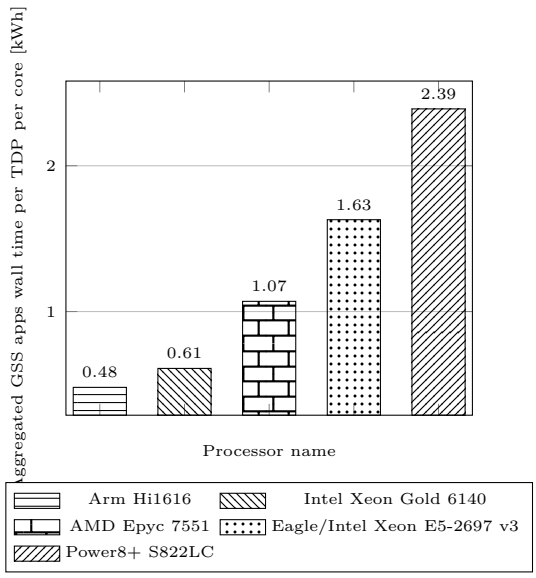
- *Cost efficiency* using scaled timing results by the cores price falling on the given number of cores (cores price is calculated by dividing processor price by a given number of processor cores).



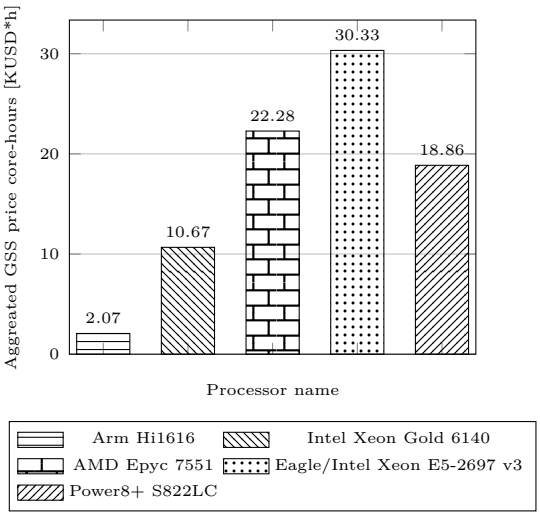
a) Ranking of architectures across the number of cores (processes)



b) Aggregated GSS benchmark walltimes



c) Ranking of architectures based on estimated energy consumption (total power needed by CPU to finish all tests) - the lower the better



d) Ranking of architectures based on cost efficiency

Figure 3. Benchmarking summary results

In the scope of *aggregated execution times* for the given number of cores, it can be noted that among the whole range of the number of cores, the winner is Intel® Xeon® Gold 6140 (Fig. 3a). Surprisingly, the AMD Epyc™ is slower than the ARM Hi1616 when 64 or more cores are used (it is mostly because hyperthreading degrades the performance in some cases) and is also slower than the reference Intel®Xeon® E5-2697 when 9 or more cores are used.

By looking at the aggregated execution time across all tested applications, the winner turned out to be Intel® Xeon® Gold 6140 (Fig. 3b).

For the *estimated cumulative energy consumption* (calculated as a sum of walltimes and TDP per used cores products expressed in kilowatt-hours) metric, the winning processor is ARM Hi1616, representing the current tendency in HPC, where attention generally speaking is turned to energy-efficient technologies. The second-place holder is Intel®Xeon® Gold 6140 and Power8+ brings up the rear (Fig. 3c).

The estimated power consumption chart is a good point of view when talking about green HPC computing. The presented results are not exact because they are only dictated by the estimated values based on the CPU's TDP. Nevertheless, assuming the fact that almost all architectures use the same memory model - DDR4, - it can be considered that most of the energy consumed is the energy utilised by the processor. In this scope, the best energy consumption ratio is characterised by the ARM architecture, which is absolutely designed for energy-saving solutions which are also widely used in mobile devices. The results for the new Intel Skylake architecture were a big surprise, which took the second place with a very similar result of about 20% more. The AMD Epyc™ demonstrated a much worse result, but from our observations, its internal architecture is better suited to applications in which I/O systems play an important role.

Another valuable finding is based on GSS cost efficiency analysis (Fig. 3d). ARM Hi1616 turned out to be approximately 9 times better than for Power8+ (equipped with DDR3, NVLinks were not utilised) and 15 times better than the reference Intel® Xeon® E5-2697 processor, mostly due to its small number of expensive cores and relatively average timing results. Additionally, when talking about general processor characteristics extracted from all the tests performed, IBM Power8+ demonstrates particularly good performance for the applications with a big number of I/O such as Pandora, OpenSWPC, CMAQ/CCTM. The best results are obtained when the total output dominates over the input and RAM consumption. In many cases, it outperforms ARM Hi1616, Intel® Xeon® E5-2697, and AMD Epyc™ for I/O intensive applications. On the other hand, Power8+ shows worse results than the above-mentioned processors if the applications are computationally extensive while producing a relatively small amount of output. This is the case of the IPF and ABMS applications. On all processors, all benchmarks show the highest efficiency if the number of MPI processes is between 2 and the total number of physical cores. After that, the efficiency usually drops remarkably as hyper threading is not utilised properly. At the same time, it is quite often that the highest speedup is reached when the number of MPI processes is significantly more than 20. It would be interesting to perform the tests on the testbeds including more nodes.

When analysed simultaneously, all the abovementioned results proved that the most promising ARM processor in the context of cost and energy consumption is the slowest one (mostly due to the low clock frequency). Other competitor, Intel® Xeon® Gold 6140, is 5 times less cost-efficient for GSS benchmark and slightly worse in energy consumption but it is approximately 3 times quicker regarding the aggregated execution time. In other words, the future HPC investors with the above information in place have the ability to decide which direction to follow: whether to reach high compute intensity, minimise costs, or try to find the golden middle.

Table 3 presents the summarised results for each individual architecture in three separate domains: walltime, energy efficiency and cost efficiency. The numbers reflect the weighted points referring to the overall applications results in a given category (in this case, the less the better). In general, the most promising processor is Intel® Xeon® Gold 6140 but those individuals for whom the cost and environmental aspects are the most important should look closely at ARM Hi1616.

Table 3. Ranking of all tested architectures (*the less the better*)

	Walltime	**Energy efficiency**	**Cost efficiency**
ARM Hi1616	1.0	0.2	0.1
Intel® Xeon® Gold 6140	0.3	0.3	0.4
AMD Epyc™ 7551	0.6	0.4	0.7
Intel® Xeon® E5-2697 v3	0.7	0.7	1.0
Power8+ S822LC	0.9	1.0	0.6

In general, it can be said, the most promising processor is Intel® Xeon® Gold 6140 but those individuals for whom the cost and environmental aspects are the most important should look closely at ARM Hi1616.

Conslusions and Future Work

The proposed benchmark gives a good evaluation tool for a relatively automatic way of proceeding with tests and receiving results which will directly allow using the best HPC architecture. It means that the end user or the resource owner may finally have different criteria to fulfil their requirements. The resource owner will focus on parameters which are globally efficient (all applications running on the machine), cost-efficient (TCO shown as CAPEX and OPEX, i.e. the investment costs vs. maintenance costs of the HPC). The end user will, however, concentrate on the fastest way to receive results and the most efficient way of parallelisation.

From that point of view, the benchmark could be extended by testing the scalability of the e-Infrastructure and the energy consumption of the running benchmark automatically. A final step of the benchmark could interpret the results for both groups of users and propose the best HPC system in terms of size and architecture (CPU, memory size, aggregated speed to external memory, if necessary).

Acknowledgements

This work has been supported by the CoeGSS (Centre of Excellence for Global System Science) project and has been partly funded by the European Commission's ICT activity of the H2020 Programme under grant agreement number: 676547.

References

1. EPA: Cmaq: The community multiscale air quality modeling system. `https://www.cmascenter.org/cmaq`, accessed: 2018-09-21

2. DTC: The users page on hurricane wrf. `https://dtcenter.org/HurrWRF/users`, accessed: 2019-05-30

3. AMD: AMD EPYC 7000 series processors: Leading performance for the Cloud era. https://www.amd.com/system/files/2017-06/AMD-EPYC-Data-Sheet.pdf (2019), accessed: 2019-05-28

4. Guo, X., Morales, C., Saastad, O.W., Shamakina, A., Rijks, W., Weinberg, V.: Best practice guide – ARM64. http://www.prace-ri.eu/best-practice-guide-arm64/ (2019), accessed: 2019-05-28

5. Cmaq inputs and test case data. https://www.epa.gov/cmaq/cmaq-inputs-and-test-case-data (2018), accessed: 2018-09-21

6. Yule, G.G.U.: On the methods of measuring association between two attributes. Journal of the Royal Statistical Society 75(6), 579–652 (1912)

7. Wolf, S., Fuerst, S., Geiges, A., Steudle, G.A., von Postel, J., Jaeger, C.C.: Electric mobility in view of green growth. https://globalclimateforum.org/wp-content/uploads/2018/12/GCFWorkingPaper3-2017.pdf (2017), accessed: 2019-05-28

8. Maeda, T., Takemura, S., Furumura, T.: OpenSWPC: An open-source integrated parallel simulation code for modeling seismic wave propagation in 3D heterogeneous viscoelastic media. Earth Planets Space 69(102) (2017), DOI: 10.1186/s40623-017-0687-2

9. Bryan, G.: Cm1 homepage. http://www2.mmm.ucar.edu/people/bryan/cm1, accessed: 2019-05-30

10. Maeda, T.: OpenSWPC - an open-source seismic wave propagation code. https://github.com/takuto-maeda/OpenSWPC (2018), accessed: 2018-09-17

11. Bryan, G.H.: The governing equations for CM1 (2013)

12. Rubio-Campillo, X.: C++/Python Agent-Based Modelling framework for large-scale distributed simulations. https://github.com/xrubio/pandora, accessed: 2018-09-17

13. Bicas Caldeira, A., Haug, V., Vetter, S.: IBM power systems S822LC for high performance computing: Introduction and technical overview. https://www.redbooks.ibm.com/redpapers/pdfs/redp5405.pdf (2016), accessed: 2019-05-28

14. Bicas Caldeira, A., Kahle, M.E., Saverimuthu, G., Vearner, K.: IBM power systems S822LC: Technical overview and introduction. https://www.redbooks.ibm.com/redpapers/pdfs/redp5283.pdf (2015), accessed: 2019-05-28

15. Chemel, C., Fisher, B., Kong, X., Francis, X., Sokhi, R., Good, N., Collins, W., Folberth, G.: Application of chemical transport model CMAQ to policy decisions regarding PM2.5 in the UK. Atmospheric Environment 82, 410 – 417 (2014), DOI: 10.1016/j.atmosenv.2013.10.001

16. Biswas, M.K., Bernardet, L., Ginis, I., Kwon, Y., Liu, Q., Marchok, T., Sheinin, D., Tallapragada, V., Thomas, B., Tong, M., Trahan, S., Wang, W., Yablonsky, R., Zhang, X.: Hurricane weather research and forecasting (HWRF) model: 2017 scientific documentation (2018), DOI: 10.5065/D6MK6BPR, accessed: 2019-05-28

17. Geiges, A., Wolf, S., Steudle, G., Fuerst, S.: Report of framework for prototyping of parallel agent based modelling systems. http://coegss.eu/wp-content/uploads/2018/11/D3.8.pdf (2018), accessed: 2019-05-28

18. Swarup, S., Marathe, M.V.: Generating synthetic populations for social modeling. http://people.virginia.edu/~ss7rs/synthetic_population_tutorial_2/slides.php (2017), accessed: 2019-05-28

19. CoeGSS-Project: Agent-based modelling framework for python. https://github.com/CoeGSS-Project/abm4py, accessed: 2019-05-30

20. Newburn, C.J., Abdurachmanov, D., Kaplan, L., McIntosh-Smith, S., McLean, M., Summoto, S., Van Hensbergen, E., Vergara Larrea, V.: The ARM software ecosystem: Are we there yet? https://arm-hpc.gitlab.io/presentations/SC17-Arm-Ecosystem.pdf (2017), accessed: 2019-05-28

21. Epstein, J.M., Pankajakshan, R., Hammond, R.A.: Combining computational fluid dynamics and agent-based modeling: A new approach to evacuation planning. PLOS ONE 6(5), 1–5 (2011), DOI: 10.1371/journal.pone.0020139

22. Hoefler, T., Belli, R.: Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 73:1–73:12. SC '15, ACM, New York, NY, USA (2015), DOI: 10.1145/2807591.2807644

23. Gienger, M., Gogolenko, S., Geiges, A., Kaliszan, D., Petruczynik, S., Januszewski, R., Wolniewicz, P.: Second report on provided testbed components for running services and pilots. http://coegss.eu/wp-content/uploads/2018/11/D5.8.pdf (2018), accessed: 2019-05-28

24. Gienger, M., Petruczynik, S., Januszewski, R., Kaliszan, D., Gogolenko, S., Fuerst, S., Palka, M., Ubaldi, E.: First report on provided testbed components for running services and pilots. http://coegss.eu/wp-content/uploads/2018/02/D5.7.pdf (2017), accessed: 2019-05-28

25. Wikichip: Hi1616 - HiSilicon. https://en.wikichip.org/wiki/hisilicon/hi16xx/hi1616, accessed: 2019-05-28

26. Richmond, P.: FLAME GPU: Technical report and user guide. https://media.readthedocs.org/pdf/flamegpu/latest/flamegpu.pdf (2018), accessed: 2019-05-28

27. Richmond, P., Romano, D.: Template-Driven Agent-Based Modeling and Simulation with CUDA, pp. 313–324. GPU Computing Gems Emerald Edition, Elsevier (2011), DOI: 10.1016/b978-0-12-384988-5.00021-8

28. Family 8335+03 IBM power system S822LC for high performance computing. https://ibm.co/2cMMb8B (2018), accessed: 2018-09-18

29. Gogolenko, S.: Software for agent based social simulation with raster inputs in distributed HPC environments. https://fs.hlrs.de/projects/teraflop/26thWorkshop_talks/WSSP26-24_Gogolenko.pdf (2017), accessed: 2019-05-28

30. Suleimenova, D., Bell, D., Groen, D.: A generalized simulation development approach for predicting refugee destinations. Scientific Reports 7(1), 13377 (2017), DOI: 10.1038/s41598-017-13828-9

31. Intel®Xeon®Gold 6140 processor specification. https://ark.intel.com/products/120485/Intel-Xeon-Gold-6140-Processor-24_75M-Cache-2_30-GHz (2018), accessed: 2018-09-18

DOI: 10.14529/jsfi190203

How File-access Patterns Influence the Degree of I/O Interference between Cluster Applications

Aamer Shah[1]***, Chih-Song Kuo***[2]***, Akihiro Nomura***[3]***, Satoshi Matsuoka***[4]***, Felix Wolf***[1]

On large-scale clusters, tens to hundreds of applications can simultaneously access a parallel file system, leading to contention and, in its wake, to degraded application performance. In this article, we analyze the influence of file-access patterns on the degree of interference. As it is by experience most intrusive, we focus our attention on write-write contention. We observe considerable differences among the interference potentials of several typical write patterns. In particular, we found that if one parallel program writes large output files while another one writes small checkpointing files, then the latter is slowed down when the checkpointing files are small enough and the former is vice versa. Moreover, applications with a few processes writing large output files already can significantly hinder applications with many processes from checkpointing small files. Such effects can seriously impact the runtime of real applications—up to a factor of five in one instance. Our insights and measurement techniques offer an opportunity to automatically classify the interference potential between applications and to adjust scheduling decisions accordingly.

Keywords: performance, I/O, file-access pattern, interference, benchmarking.

Introduction

The computational demand of HPC applications is continuously growing, raising the performance expectations of cluster users to unprecedented levels. In order to accommodate such demands, HPC systems frequently employ specialized designs such as multi-dimensional torus networks, GPU-based accelerators, and powerful parallel file systems. The latter are needed to provide service for an enormous amount of file accesses in parallel. Such parallel file systems are installed as centralized resources with a middle layer of I/O servers connected to storage devices at one end and to compute nodes at the other. Decoupling compute resources from I/O resources allows for better management and scalability of the I/O subsystem. However, the centralized design also means that multiple applications may share the same file system. This can lead to contention in the event of simultaneous file access and can substantially degrade application performance. Applications that perform frequent file access requests or access massive amounts of data are especially sensitive to such conditions, adding an element of variability to their performance [36].

HPC applications that perform frequent or massive file access requests are quite common. Examples include data-intensive codes such as MADCAP cosmic microwave background analyzer [34] and GCRM global cloud system resolving model [40]. They both write massive amounts of data during execution, resulting in numerous write requests. In contrast, OpenFOAM continuum mechanics solver [17] and Community Atmosphere Model (CAM) [22] of the Community Earth System Model (CESM) [33] frequently checkpoint their state, resulting in small but recurring writes. Overall, very different classes of file-access patterns can be distinguished. Not only do these patterns access the file system in unique ways, but their sensitivity to interference

[1]Laboratory for Parallel Programming, Technische Universität Darmstadt, Darmstadt, Germany
[2]Taiwan Semiconductor Manufacturing Company Limited, Hsinchu, Taiwan
[3]Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo, Japan
[4]Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo, Japan

from other applications that access the file system at the same time also varies widely. Likewise, they actively interfere with other I/O-intensive applications in different ways. All this makes access patterns to be an important factor for file-system contention. Our initial experiments with different access patterns revealed negligible interference in the case of read-read and read-write contention. These results are also consistent with word-of-the-mouth understanding in the HPC community. Therefore, we concentrated our investigation on write-write contention and the most common access patterns involved.

File system contention and the associated performance degradation are well-known [8]. In this context, the influence of request size and process count has already been studied from a single-application perspective [21], while process count has been identified as a factor of dominance when two applications compete for the file system [29]. Similarly, file-access patterns have been studied in various contexts [12, 14, 20, 25, 37]. The novelty of our research is that we study common write patterns found in HPC applications from the perspective of simultaneous access from different applications. To this end, we first developed a micro-benchmark capable of producing three distinct file-access patterns, simulating those of real applications. Two of these patterns mimic application checkpointing and out-of-core processing, while the third pattern mimics writing large files. We explored the interference potential of these patterns by running them simultaneously against each other, in the form of either micro-benchmarks or realistic applications covering check-point-intensive and data-intensive access patterns. We not only observed different levels of interference between different patterns, but also identified some general rules such as writing large output files dominating checkpointing at smaller checkpoint sizes, with the trend being reversed for larger checkpoint sizes.

In our previous work, we analyzed write access patterns and their effect on interference [6]. In this work, we expand on the topic with a more realistic checkpointing pattern, evaluate how the interference potential depends on the number of processes the application runs with, and confirm our findings with a larger set of production codes. We summarize our contributions as follows:

- An experiment design that allows the quantification of interference between different file-access patterns.
- An I/O-server monitoring capability added to the hitherto purely application-centric interference profiler LWM^2 [9], enabling us to isolate distinct interference phenomena even in noisy environments.
- An analysis of the interference potential of common file write patterns in HPC applications, including the identification of a typical combination with high interference potential.

Taken together, our results pave the way for an effective reduction of interference in the future. Specifically, it brings us much closer to the automatic recognition of applications with high interference potential, allowing their I/O to be separated either in space or time.

The remainder of the paper is organized as follows. First, we provide the necessary background information on parallel file systems in Section 1. In Section 2, we present our approach, including a taxonomy of file-access patterns, an explanation of our experiment design, an introduction to the interference profiler LWM^2, and a description of the I/O server monitor added to LWM^2 for the purpose of this study. After that, we present our results in Section 3, ranging from micro-benchmark-only experiments to measurements with realistic applications. Finally, we review related work in Section 4 before we draw our conclusions and outline future perspectives in Conclusion section.

1. Parallel File Systems

In order to accommodate an increasing number of concurrent file accesses, cluster file systems evolved from a simple client-server model in the style of NFS into usually dedicated clusters of servers and storage devices called parallel file systems. In the most common configuration, a parallel file system connects servers and storage devices via a dedicated network, while it connects servers to compute nodes via a shared message-passing network, as shown in Fig. 1. Clients running on compute nodes forward file-access requests to the I/O servers. Then I/O servers then distribute them to the attached storage devices—according to the mapping of files onto storage devices. This allows handling simultaneous file accesses with better performance. Additionally, striping individual files across multiple storage devices supports efficient parallel access to a single file. Following these general design principles, several implementations such as Lustre, GPFS, FhGPS, PVFS, PanFS, and HDFS emerged. Below, we describe two popular parallel file systems used in our experiments in more details.

1.1. Lustre

Lustre is a file-storage system for clusters used by many of the Top500 HPC systems [24]. It offers up to petabytes of storage capacity and provides multiple gigabytes per second of I/O throughput. Its architecture distinguishes two basic types of servers: *metadata servers* (MDSs) and *object storage servers* (OSSs), as shown in Fig. 1. An MDS stores file-system structure information, including directory layout and file attributes. An OSS stores the actual file-data stripes on the attached *object storage targets* (OSTs). When an I/O request is made, MDS and OSS internally perform different types of file accesses. The MDS performs search and small read and write operations on the file structure information, while the OSS performs potentially large reads and writes on the actual file. Decoupling metadata from data makes it possible to optimize each server type for its most frequent access pattern.

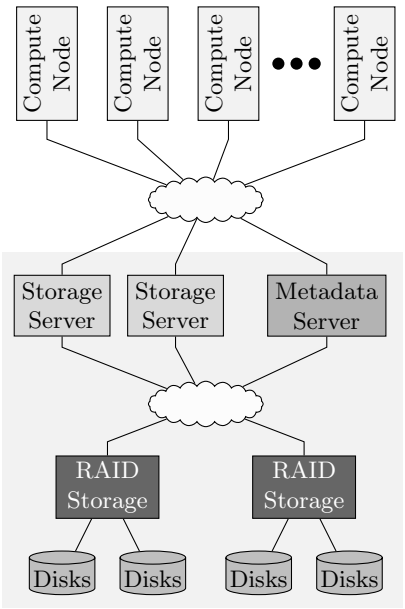


Figure 1. A typical Lustre configuration, with separate I/O servers for metadata and file storage

1.2. GPFS

General Parallel File System (GPFS) is a proprietary parallel file system developed by IBM [3]. It is often found on Blue Gene systems but is also available on other HPC clusters such as TSUBAME 2.5. It supports multiple configurations, including the shared-disk-cluster configuration, in which every compute node manages a part of the file system. However, on large HPC systems, a separate I/O subsystem is more common. In such a configuration, GPFS can span an I/O subsystem with thousands of nodes. GPFS stores data files and their associated metadata on the same block-based devices called *network shared disks* (NSDs). This makes GPFS also suitable for applications with small file accesses such as Web servers. GPFS stripes data files across all disks in a storage pool, achieving high performance. In addition, internal storage pools can be defined to provide different levels of availability and performance for certain files.

2. Approach

Many HPC applications are data-intensive, that is, they perform extensive I/O operations. They employ different I/O libraries and file formats and produce different process-to-file ratios. Because of the fact that a significant proportion of applications still use POSIX-IO or MPI-IO in the classic one-file-per-process manner [15], we concentrate our experiments on this configuration, while also evaluating MPI shared-file scenarios. Given that using MPI-IO with one file per process is essentially equivalent to POSIX-IO [34], at least on our test systems and on many others, our micro-benchmarks exercise only MPI-IO.

Algorithm 1 Open-Write-Close

```
loop
    Open New File
    Write chunksize
    Flush I/O Writes
    Close File
end loop
```

Algorithm 2 Write-Seek

```
Open File
loop
    Seek to the beginning
    Write chunksize
    Flush I/O Writes
end loop
Close File
```

Algorithm 3 Aggregate-Write

```
Open File
loop
    Write chunksize
end loop
Close File
```

Figure 2. Three I/O access patterns

2.1. File Access Patterns

HPC applications exhibit a variety of file access patterns, whose frequent checkpointing, file accesses for out-of-core processing, and writing of large output files are considered here. We implemented three characteristic patterns corresponding to these three use cases as micro-benchmarks, ran them with a range of file sizes, and measured their interference potential when executed against each other as well as against realistic applications. Figure 2 shows the pseudo-code of the three patterns.

Open-write-close. The first considered pattern we consider is called open-write-close (OWC) (Listing 1). In this pattern, each process creates a new file, writes data to it and then closes it. In the next iteration, a new file is created again for data writing. The pattern is commonly used for checkpointing in many applications, such as Flash [30], CESM [33] and OpenFOAM [17]. This access pattern generates a large number of metadata operations, while the actual amount of data written to files can be small. On systems with limited metadata resources such patterns can quickly create a bottleneck at scale. Compared to our previous work [6], we have updated the open-write-close pattern to create a new file in each iteration, mimicking the checkpointing pattern in a more realistic fashion.

Write-seek. In the write-seek (WS) pattern (Listing 2), a process opens a file at the beginning. It then writes a chunk of data to it, and then seeks back to the beginning of the file. At the end of execution, the process closes the file. This pattern is similar to the open-write-close pattern in the sense that it performs a massive number of small file accesses. However, it generates less metadata traffic as it reuses the same file, keeping it continuously open. Between individual writes, only seek operations take place. The write-seek pattern captures a simplified version of file accesses during out-of-core processing of HPC applications, such as in MADCAP [28]. Facing memory capacity pressure, HPC applications often have to resort to out-of-core processing. This means they write data they cannot hold in the main memory temporarily to a file, and read it back once it needs to be processed. This results in a write-seek-read pattern. The pattern can have many different instantiations with respect to write size, seek size, and read size. For simplicity as our goal is measuring write-write interference potential, we have reduced the pattern to a write followed by a complete seek.

Aggregate-write. In the aggregate-write (AW) pattern (Listing 3), a process opens a file at the beginning and then continues to append chunks of data to it. The file gets closed at the end of execution. This pattern is similar to large writes in such applications as MADCAP [34] and GCRM [40]. The pattern involves a few metadata operations but many write operations, resulting in large file sizes. At scale, this pattern can substantially challenge the performance of an I/O subsystem.

Client-side I/O caching requires flushing the I/O traffic after every write operation for the open-write-close and the write-seek patterns. Otherwise, writes of small chunks remain cached in buffers for each OST in the Lustre client and are overwritten with the next write. We have also found flushing of write buffers in real applications to be a common practice. Therefore, our addition of buffer flushes is not unusual. The need for flushes does not arise for writes of large chunks if the chunk size is larger than the OST buffer size. Moreover, this issue does not affect aggregate-write, in which small writes are initially collected in the OST buffer and eventually are committed to the file system. In order to have a consistent benchmark, writes were flushed for both Lustre and GPFS, and for all chunk sizes.

2.2. Capturing Interference

To capture incidents of interference, we run the patterns side by side and measure the change in throughput in comparison to an isolated run. We call the benchmark whose throughput degradation we are interested in the *probe*. The throughput degradation serves as a quantification of the *passive* interference it suffers. The benchmark causing this degradation through *active*

interference is called the *signal*. To study the way that interference effects evolve over the runtime of a specific, more complex probe application, we let the signal benchmark to also produce its pattern in a *periodic* fashion, with I/O activity being interrupted by silent phases without the I/O activity. Whenever the signal shows activity, the probe may suffer a dent whose depth indicates the severity of the interference.

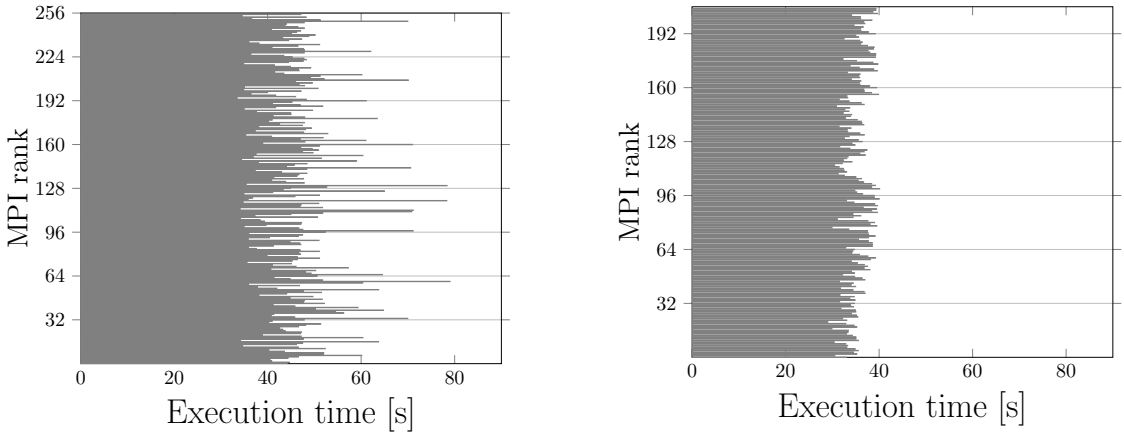
In order to measure how the I/O throughput of an application changes, we use the profiler LWM^2 [9] after extending it to suit our requirements. LWM^2 is a lightweight profiler designed to collect the most basic performance metrics with as little overhead as possible. The I/O metrics relevant to our study are all measured in dynamically loaded interposition wrappers. One aspect important to our study is the ability of LWM^2 to represent performance dynamics in *time slices*. In addition to production of a compact performance summary covering the entire runtime, LWM^2 splits the execution into fixed-length time slices and generates a profile for each of them. The time slice boundaries are synchronized across the entire system by aligning them with the system time. As a result, the simultaneity of performance phenomena occurring in different applications can be easily established. This is useful because it may indicate a causal relationship between these phenomena. The duration of time slices is configurable. In our experiments, we use a time-slice length of 4 seconds and a period length of 24 seconds for the periodic version of our micro-benchmarks. In this way, each period covers at least a few time slices.

However, the mostly application-centric perspective of LWM^2 confronts us with two challenges: noise from other applications not related to our experiments and irregular behaviors of the I/O servers themselves. Ideally, I/O interference experiments should be conducted in a fully controlled, noise-free environment. In practice, however, reserving an entire production cluster for an extended period of time is too expensive. Moreover, the throughput delivered by I/O servers is often non-uniform. For example, the exhaustion of cache space may result in a sudden throughput drop. As a consequence, such irregularities may further blur the interference effects we want to study.

In order to be able to keep our measurements as clean as possible from these two effects, we extended LWM^2 to monitor activities of the I/O server during execution of an application as well. The server activities are captured in every time slice, allowing us to correlate events across applications and I/O servers. In particular, this allows runs to be filtered out where the file-server load is 10% higher than the application I/O traffic captured by POSIX/MPI-IO wrappers. In addition, server-side monitoring allowed us to learn more about certain non-uniform but to some degree predicable behaviors, which we are now able to exclude from our measurements, as explained in Section 2.3. For both GPFS and Lustre, we estimated the I/O traffic to and from the servers by profiling the InfiniBand counters of the servers. Moreover, for Lustre, we parsed the diagnostic data updated by the Lustre client software running on each node to capture the amount of reads and writes from/to the I/O servers.

2.3. Server-side Imbalance

In some experiments, we observed substantial differences among the execution times of individual processes of an application that occurred sporadically with both file systems. In such cases, most processes finished within the expected time, while the remaining ones had to keep performing I/O for a significantly longer duration, sometimes more than twice as long, as shown in Fig. 3a. Such observations are not uncommon and have been reported before [5]. One major factor revealed in a closer investigation of the imbalance effect was unbalanced load on the file-



a) Default

b) 1 file / OST

Figure 3. Mapping one file to one OST reduces the runtime imbalance among processes

server side, as shown in Fig. 4. In particular, this happened with Lustre, where files are randomly assigned to an OST in the one-file-per-process mode.

When an OST was shared by many processes, its performance dropped, which in turn affected the throughput of the associated I/O server. We confirmed this observation by artificially enforcing an equal number of files per OST in a small experimental run, which reduced the disparity of execution time by more than 75%, as shown in Fig. 3b. However, such enforcement is not feasible in a real world scenario, as it requires the number of processes to be a multiple of the number of OSTs. With GPFS, the process imbalance effect occurred to a lesser extent with large files because they were automatically striped across all NSDs, but more predominantly with small files below the stripe size presumably for the opposite reason. Besides the OST/NSD load imbalance, other factors, such as the straggler phenomenon [5], might also contribute to the imbalance.

To accommodate the variance resulting from this imbalance, while still being able to discern interference effects, we considered only the balanced part of a run. This approach is justifiable since the imbalance only affects the later stage of a run, in which only a small portion of the total I/O volume is written. In practice, we found that the I/O traffic in this tail-off stage is usually less than 10%. As a result, we calculated the throughput drop and runtime dilation, our comparison metrics, only up to the moment when the first of the two simultaneously running programs had written 90% of its data volume. Even though this empirical technique did not completely remove the effects of the server-side imbalance, it reduced the resulting imprecision significantly and consistently, while preserving the effect of interference.

3. Evaluation

This section presents the results of our interference experiments. In these experiments, we first ran pairs of our micro-benchmarks against each other to study the interaction of the different patterns in their purest form. To confirm our findings, we then executed the micro-benchmarks against three realistic applications, OpenFOAM, MADbench2, and HACCIO, used for simulations of fluid dynamics, cosmic background radiation, and collisionless cosmic fluid creation, respectively. Finally, we analyzed the interference effect observable between two instances of each of these applications.

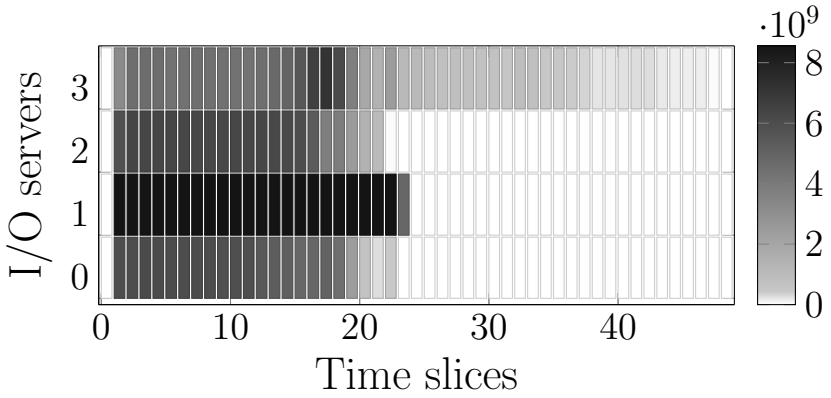


Figure 4. Write throughput of I/O servers. The performance of server 3 is degraded, leading to a longer execution time of I/O operations and hence the application

3.1. Environment

The results were obtained on the TSUBAME 2.5 supercomputer hosted at Tokyo Institute of Technology, Japan. The cluster comprises nodes in different configurations. The nodes used in our experiments make up the majority of the cluster and are equipped with two Intel Xeon X5670 (Westmere-EP, 2.93 GHz) 6-core processors, three NVIDIA Tesla K20X (GK110) GPUs, and 58 GiB DDR3 main memory. The cluster employs a two-rail fat-tree InfiniBand 4X QDR network, used both for message passing and file I/O traffic. The peak performance of the cluster is 2843 TFLOPS.

TSUBAME 2.5 offers GPFS and Lustre file systems for parallel I/O at different mount points, which are frequently updated. The configuration used in our experiments is as follows. GPFS on /data0 is hosted on four file servers (NSD servers), each connected to 14 RAID storage devices (NSDs), while Lustre on /work1 is hosted on eight file servers (OSSs), each of them connected to 13 RAID storage targets (OSTs). We used only these mount points in our experiments. On Lustre, metadata requests are handled by one MDS server with one additional standby server. The `qos_threshold_rr` parameter of Lustre has been set to 16%, meaning that storages are selected mostly in a round robin fashion. Additionally, TSUBAME 2.5 also provides 120 GB SSDs on compute nodes as scratch space. All file servers are equipped with two InfiniBand 4X QDR adapters, connecting them to one of the two rails of the fat-tree network. Table 1 provides a summary of the two file systems on the mount points we used.

Table 1. Specifications of the file systems on TSUBAME 2.5 used in our experiments

PFS	Mount point	Metadata server	File server	disks per server	Bandwidth
GPFS	/data0	N/A	4	14	20 GB/s
Lustre	/work1	1	8	13	50 GB/s

3.2. Experimental Setup

Except for the experiments comparing patterns at different process counts, a single instance of a mirco-benchmark or an application consisted of 256 processes, utilizing 64 compute nodes.

As the experiments were carried out on a production system, we took care of filtering out runs with more than 10% external noise. The filtering was done using the I/O server monitoring module of LWM^2. We also repeated each experiment five times and took the best-performing run (i.e., with the lowest degree of external interference).

We executed patterns with file sizes ranging from 1 MiB to 256 MiB on a logarithmic scale. For the open-write-close pattern and write-seek pattern, this meant that a file of the specified size was written repeatedly, while for the aggregate-write pattern this meant that each write operation had the specified buffer size.

3.3. Micro-benchmarks

In order to understand the interaction of different I/O access behaviors, we first paired up the three access patterns to form a collection of interference scenarios. We ran each of the three patterns against itself and against the other two, resulting in six experiments. For the purpose of interference quantification, however, we had to consider each micro-benchmark once as a signal and once as a probe, resulting in a total number of nine scenarios (i.e., $\{OWC, WS, AW\}^2$).

Table 2. Write bandwidth observed in an experimental run on Lustre when probe open-write-close is exposed to three different signal patterns at a chunk size of 1 MiB

		Signal		
		Open-write-close	Write-seek	Aggregate-write
Standalone	Bandwidth [GB/s]	28.4	31.6	42.2
Signal	Bandwidth [GB/s]	16.1	19.8	3.8
	Degradation [%]	43.31	35.76	9.95
Probe	Bandwidth [GB/s]	16.3	16.8	9.6
	Degradation [%]	42.61	40.85	66.2

Table 2 shows the write throughput when an open-write-close probe is exposed to three different signal patterns. For both the probe and the signal patterns, we show the standalone and the interfered throughput. We also quantify severity of the interference effect in terms of the percentage degradation of the throughput T, defined as:

$$T = \frac{T_{standalone} - T_{interfered}}{T_{standalone}} \times 100.$$

A high value of the degradation indicates severe interference inflicted by the signal pattern. As the focus of this paper is the severity of the interference, we restrict ourselves to relative throughput degradation figures in the remainder of the paper.

3.3.1. Access Patterns

We executed the complete set of combinations on both GPFS and Lustre for chunk sizes of 1 MiB, 16 MiB, and 256 MiB. Figure 5a shows a throughput drop observed with all pattern combinations, for chunk sizes of both 1 MiB, 16 MiB, and 256 MiB. With the smaller chunk sizes, we found aggregate-write to have a clearly higher interference potential than the other two

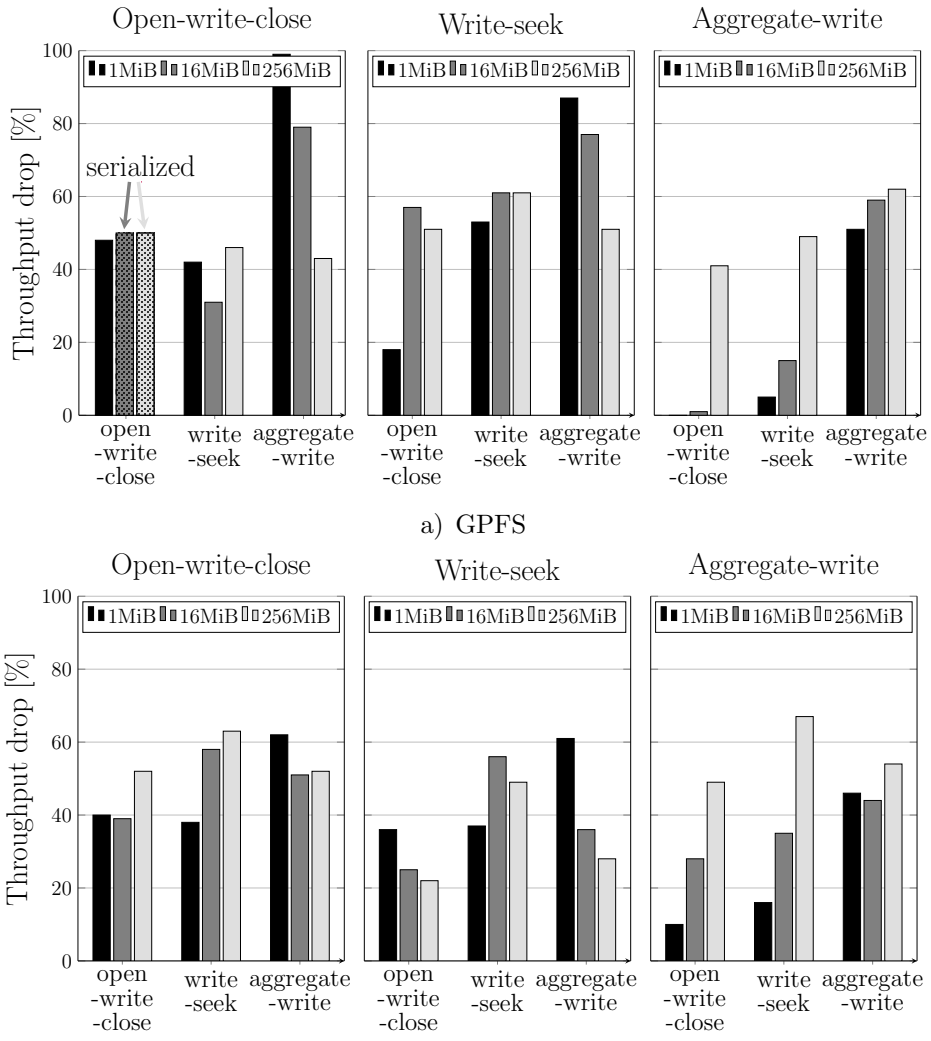
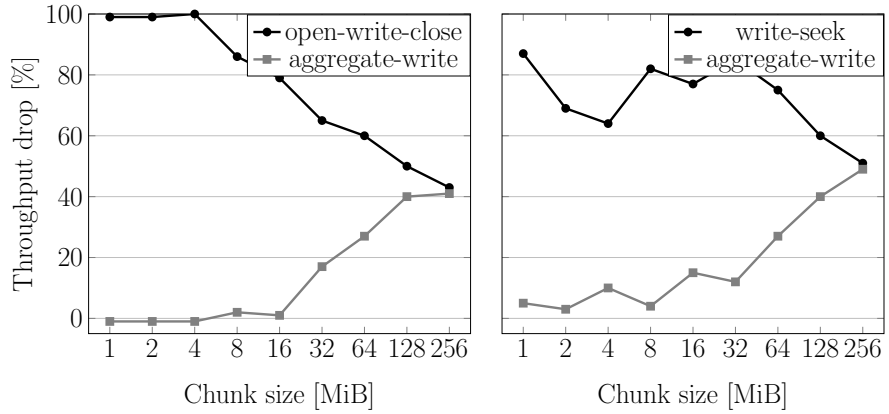


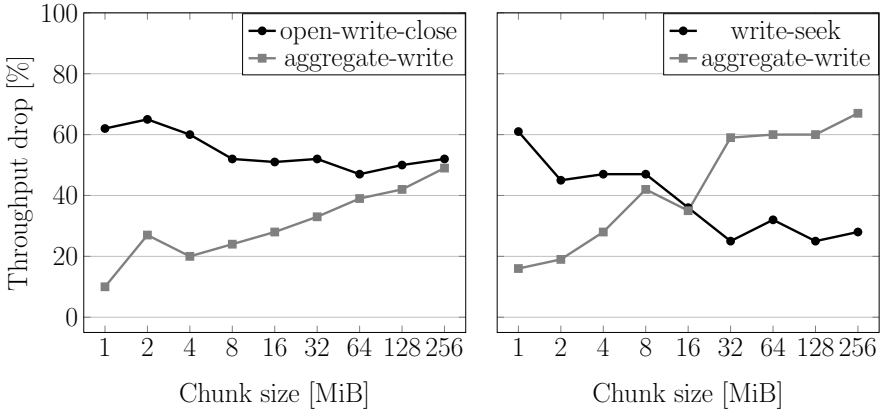
Figure 5. Throughput drop when the patterns are executed against each other. A higher bar means lesser throughput and higher passive interference. The top patterns indicate the probe, while the patterns below the x-axis indicate the signal

patterns. When open-write-close and write-seek are executed against each other, their throughput drops by about 45% to 55%. This can be explained by the equal sharing of I/O resources between them. However, concurrent execution of aggregate-write against the other two patterns reduces the latter's throughput by more than 80%, while the effect of the two other patterns on aggregate-write itself is much smaller. This indicates that aggregate-write dominates these two patterns at a chunk size of 1 MiB and 16 MiB, occupying most of the I/O resources. At a chunk size of 256 MiB, the I/O resources are distributed more evenly among the patterns. It can be seen that open-write-close is less affected by aggregate-write compared to the 1 MiB case, while aggregate-write is more affected by the other two patterns. Open-write-close, at chunk sizes 16 MiB and 256 MiB, when executed against itself, becomes serialized, that is, one pattern of the pair executes first, almost completely degrading the second pattern during first's execution.

We repeated the same set of experiments on Lustre. The results from the nine pair-wise combinations of patterns for 1 MiB, 16 MiB, and 256 MiB are shown in Fig. 5b. The general trend of the interference potential for the three chunk sizes is the same as on GPFS but with different intensities. At a chunk size of 1 MiB, aggregate-write generates most of the interference,



a) GPFS



b) Lustre

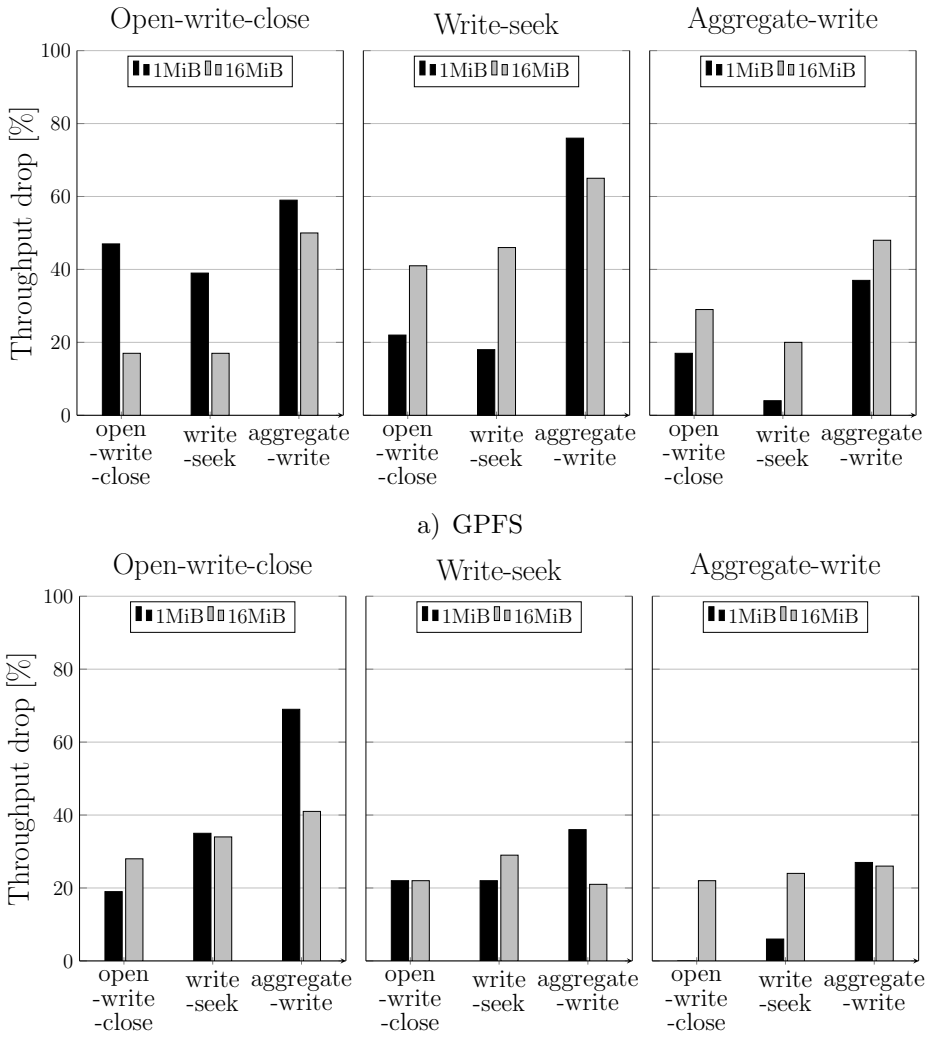
Figure 6. Effect of chunk size on throughput degradation

again while itself being the least affected one. However, the disparity is not as strong as on GPFS. As the chunk size is increased to 16 MiB and 256 MiB, respectively, the interference potential of aggregate-write decreases, whereas that of open-write-close and write-seek increases. At a chunk size of 256 MiB, write-seek causes most of the throughput reduction, more than 60% for the other two patterns.

3.3.2. Chunk Size

As chunk size seems to be a crucial parameter for the interference potential of the above mentioned patterns, we investigated this more closely by running open-write-close and write-seek against aggregate-write for chunk sizes ranging from 1 MiB to 256 MiB on a logarithmic scale. The results for GPFS are shown in Fig. 6a. Open-write-close seems to share I/O resources with aggregate-write more evenly as the chunk size increases, with 256 MiB being the break-even point. Write-seek shows a similar trend but with the slope shifted to the right. The convergence here begins when a chunk size of 32 MiB is reached. Beyond this point, the progression is similar to open-write-close, as I/O resources start to be shared more evenly. At the last data point of 256 MiB, the two patterns break even.

We also studied the sensitivity of interference to chunk sizes on Lustre. The results are summarized in Fig. 6b. The trend of open-write-close on GPFS, where, at small chunk sizes, aggregate-write dominates over open-write-close, reappears on Lustre. As the chunk size in-



a) GPFS

b) Lustre

Figure 7. Throughput drop when the patterns are executed against each other. A higher bar means less throughput and a higher passive interference. The pattern above each chart represents the probe, whereas the patterns below the x-axis represent the signal. The signal pattern is executed in a periodic fashion

creases, open-write-close starts to perform better. The trend culminates at 256 MiB, where open-write-close and aggregate-write experience the same amount of throughput drop. In the case of write-seek, aggregate-write dominates at small chunk sizes. However, as the chunk size is increased, the trend is quickly reversed. Both patterns suffer the same amount of throughput degradation at 16 MiB, beyond which write-seek starts to dominate aggregate-write.

3.3.3. High Frequency vs. Low Frequency

As the sensitivity to chunk size shows, the trend of interference among the patterns depends on their specific characteristics. To evaluate this further, we consider the file access frequency of a pattern. However, covering the whole breadth of possible write access frequencies is prohibitively expensive. Instead, we expose the unaltered probe to a periodic signal, in which write activity phases alternate with computational busy-wait phases, mimicking bursty I/O. The period length of the signal was set to 24 seconds so that multiple consecutive time slices of LWM^2 fall in one

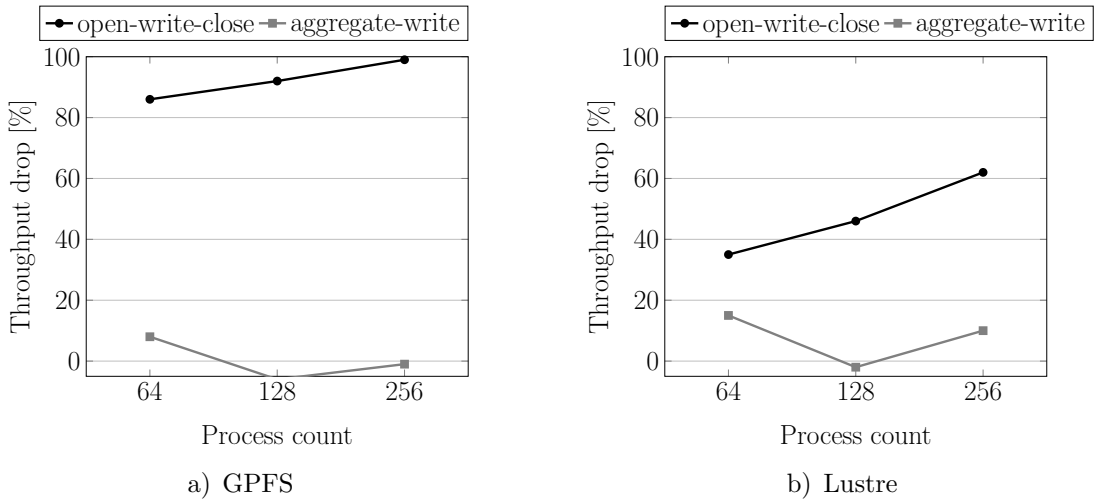


Figure 8. Effect of process count on the passive degradation produced by patterns on GPFS and Lustre

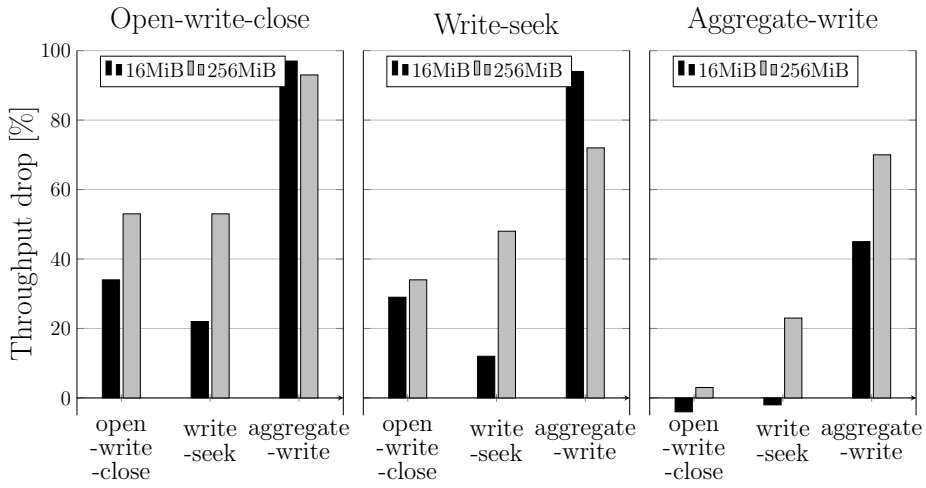
period. Note that the write activity phases are repeated multiple times in a run. The results of the experiments are shown in Fig. 7.

Overall, the interference trend, both for GPFS and Lustre, is similar to what was observed for the sustained activity patterns, but with a lower intensity. Aggregate-write still dominates over open-write-close and write-seek at a chunk size of 1 MiB. Similarly, at the larger 16 MiB chunk size, aggregate-write causes a lesser degradation while finding itself being victimized to a higher degree. On GPFS, at a chunk size of 16 MiB, open-write-close suffers less throughput degradation against itself and against write-seek. One of the reasons it can be like this is at lower frequencies and at larger write chunk size, is that the metadata operations cease to be the I/O bottleneck, while, additionally, the low frequency prevents the write bandwidth of the system from being saturated. As a result, the performance of open-write-close degrades to a lesser degree.

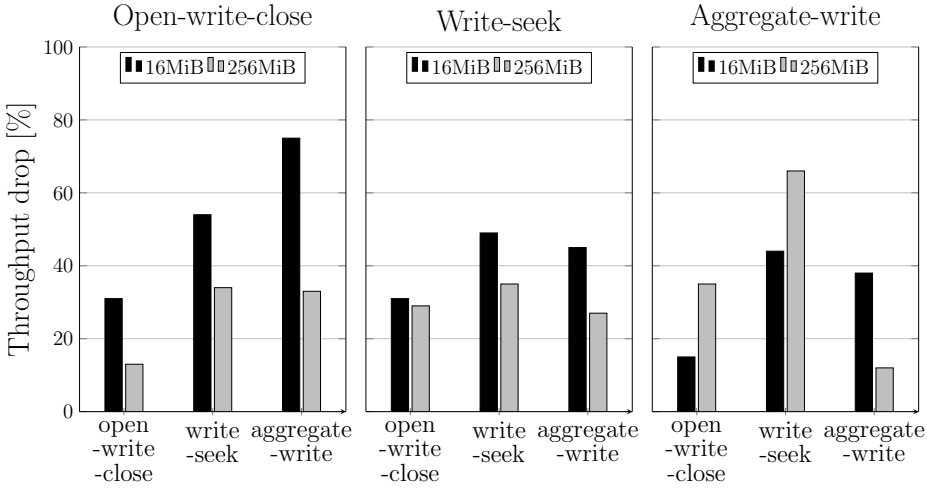
3.3.4. Process Count

It has been previously observed that an application with higher process count dominantly occupies the I/O resources of a system when run against an application with lower process count [29]. However, does this relationship hold true if the two applications have different file access patterns? We investigated this by running open-write-close and aggregate-write against each other with a chunk size of 1 MiB. Because write-seek is similarly dominated by aggregate-write at this chunk size, we have concentrated our study on open-write-close. For each run, we executed the open-write-close pattern with 256 processes and the aggregate-write pattern with 64, 128 and 256 processes. The results of the experiments are shown in Fig. 8.

The blue line shows the throughput degradation of open-write-close while the red line shows the throughput degradation of aggregate-write. For GPFS, we see in Fig. 8a that open-write-close is degraded severely, even when aggregate-write occupies only one fourth of the space. As we increase the process count of aggregate-write, open-write-close degrades even more severely. Figure 8b shows the trend for Lustre, which is similar to that of GPFS, but with a lesser degradation. Again, we see that open-write-close suffers higher degrees of degradation when run against aggregate-write, even when aggregate-write is one fourth of the size. The throughput



a) GPFS



b) Lustre

Figure 9. Throughput drop in MPI shared-file mode when the patterns are executed against each other. A higher bar means less throughput and higher passive interference. The pattern above each chart represents the probe, whereas the patterns below the x-axis represent the signal

of open-write-close decreases even further as the process count of aggregate-write increases. From these experiments we can conclude that, when it comes to sharing I/O resources between applications, the write-access pattern can play a bigger role than the application process count.

3.3.5. Shared File

Not to ignore this increasingly common mode, we also performed a set of experiments on shared files. The file was shared in such a way that each process occupied a contiguous portion of the file. For open-write-close and write-seek, the size of the contiguous portion exactly matched the chunk size of the benchmark. For the aggregate-write pattern, the contiguous portion matched the size of the total data written by a process.

Figure 9a shows the interference potentials on GPFS. At a chunk size of 16 MiB, aggregate-write dominates the other two patterns significantly, while at 256 MiB, even though being still dominant, it generates comparatively less interference for other patterns. These observations are similar to the results with one file per process. However, we observed some cases in the pairwise

execution of the patterns, in which one instance would completely dominate over the other instance, effectively serializing the I/O traffic between the pairs. This near-serialization of the I/O traffic was observed when running patterns against themselves as well as against different patterns. Figure 9b presents the results on Lustre, where we observed that aggregate-write dominates open-write-close at a chunk size of 16 MiB, while becoming slightly dominated by write-seek itself. At a chunk sizeof 256 MiB, open-write-close and write-seek are evenly interfered in all the runs but dominate aggregate-write. The behavior of aggregate-write is again consistent with the one-file-per-process case. Open-write-close, however, is less prone to interference at larger chunk sizes. Based on these observations, and considering that writing shared files is a topic of research in its own right with its own characteristic set of access patterns, we believe that a full coverage of shared files would justify a separate study.

3.3.6. Discussion

From the above results, it is clear that different I/O access patterns show different interference potential. The chunk size is also an important factor in determining which pattern is dominant. At smaller chunk sizes, aggregate-write prevails over open-write-close and write-seek, causing a notable degradation of throughput for the latter two while showing little impact on the former. However, as the chunk size increases, the balance is shifted in favor of open-write-close and write-seek. At a certain point, open-write-close and write-seek suffer as much as aggregate write, beyond which the trend may even become reversed. On GPFS, open-write-close and write-seek show similar degradation trends, while on Lustre, open-write-close has comparatively less interference potential. The precise reason for our observations is unclear, but it seems that both metadata operations including open, close, and seek on the one hand and the number of different file blocks an application writes make it sensitive for interferention. At least, this would explain the trend reversal shown in Fig. 6b. As the chunk size, increases together with it the number of different blocks written by aggregate-write, the density of metadata operations shrinks.

3.4. Applications

After establishing an interference relationship among the access patterns through micro-benchmarks, we investigated the same effects using realistic applications. First, we verified the interference trend for micro-benchmarks against applications, and later confirmed it for application vs application. In our previous study [6], we considered two typical I/O-intensive HPC applications, OpenFOAM and MadBench2. Here, we extend the work by also evaluating our approach with HACCIO, a code-writing large checkpoints, against the micro-benchmarks and the other two applications. All three together of them provide one realistic use for each of the three access patterns, as summarized in Tab. 3.

Table 3. Applications and the access pattern they represent including the chunk size

Application	Access pattern	Chunk size
OpenFOAM	open-write-close	a few kilobytes
MADBench2	write-seek	74 MB
HACCIO	aggregate-write	386 MB

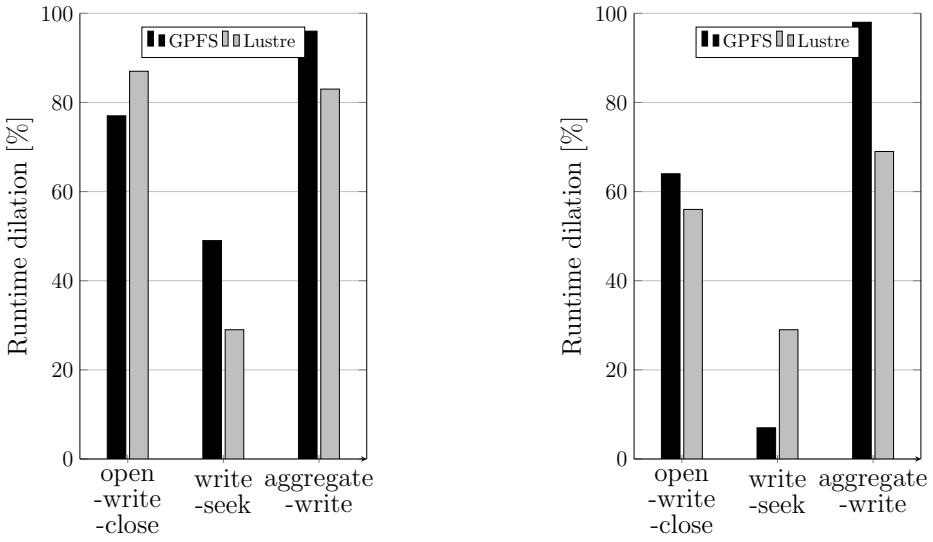


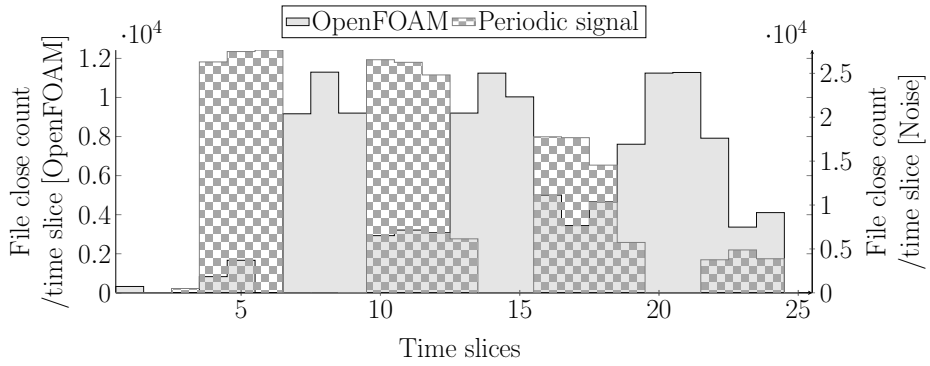
Figure 10. Throughput degradation of OpenFOAM when run against the patterns at different chunk sizes

3.4.1. OpenFOAM

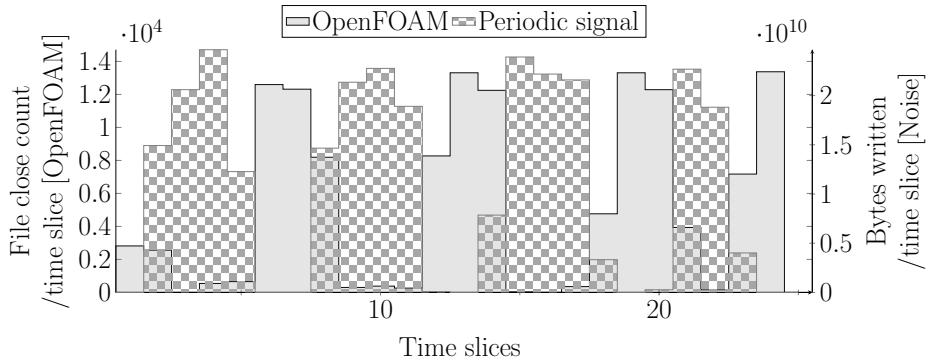
OpenFOAM [17], which stands for Open source Field Operation And Manipulation, is a free, open source computational-fluid-dynamics (CFD) software package developed by OpenCFD Ltd of ESI Group and distributed by the OpenFOAM Foundation. It was one of the first scientific applications to leverage C++ for a modular design. The package provides parallel implementations of a rich set of libraries, from mathematical equation solvers to general physical models. OpenFOAM uses standard C++ I/O for checkpointing at regular intervals. At each checkpoint, new files of around a few kilobytes are created and written by every process, making its I/O behavior similar to the open-write-close pattern. As LWM^2's C++ I/O profiling is still in progress, we were only able to capture file-close counts for our runs. In our experiments, OpenFOAM closed more than 14000 files per time slice. As this count is significantly larger than the process count of the application, it indicates that most of those files were written to and closed in the same time slice, making the file-close count an indicator of I/O throughput. Similarly, we used the dilation of execution time, which occurs as a consequence of I/O performance drop, to gauge the interference potential.

In our experiment, we ran the cavity example from the official tutorial of OpenFOAM version 2.3.0 using 256 processes. Cavity involves processing of an isothermal, incompressible flow in a two-dimensional square domain. Specifically, we used the icoFoam solver, in which the flow is assumed to be laminar. We executed the cavity example in parallel with each of the three pattern micro-benchmarks. We set the chunk size of the patterns to 1 MiB, the smallest chunk size we used in our pure micro-benchmark experiments. We executed the runs on both GPFS and Lustre, and adjusted the runtime of the patterns to fully overlap with OpenFOAM's execution.

OpenFOAM experienced degraded I/O performance when executed concurrently with all the three patterns. The throughput drop caused by each of the patterns is shown in Fig. 10a. As OpenFOAM's I/O pattern is similar to open-write-close with a small chunk size, the large interference potential of aggregate-write at such a small chunk size is immediately visible, leading



a) OpenFOAM vs. open-write-close



b) OpenFOAM vs. aggregate-write

Figure 11. Time-slice view of OpenFOAM when executed concurrently with two patterns on GPFS

to more than 80% drop for Lustre and 90% for GPFS. Unlike the micro-benchmark, OpenFoam also suffered about 80% throughput drop against open-write-close. One reason for this behavior might be the unequal chunk size of the patterns and OpenFOAM. To verify this, we executed the patterns at 2 KiB chunk size. The results are shown in Fig. 10b. Aggregate-write still dominates over OpenFOAM, with a throughput drop of more than 90% for GPFS. Lustre, on the other hand, shows a slightly reduced drop of 70% in throughput. Open-write-close now degrades OpenFOAM's throughput by around 60%, similar to what the micro-benchmark allowed us to see. The interference of write-seek on Lustre remains at 30% for both chunk sizes. However, it declines from around 50% for 1 MiB to 10% for 2 KiB on GPFS. Overall, the interference trend is similar to that of our purely micro-benchmark-based observations.

To further understand the I/O interference dynamics during concurrent execution, we executed OpenFOAM against periodic modes of open-write-close and write-seek. In this mode, the micro-benchmark's I/O access phases alternate with silence. This periodic mode highlights the effects of interference during the I/O access phases. Figure 11a and Figure 11b show the *time slice view* when OpenFOAM is concurrently executed with open-write-close and aggregate-write, respectively. Against open-write-close, OpenFOAM's performance degrades by 60%–70% during active phases of the pattern in comparison to the silent phases. On the other hand, OpenFOAM against aggregate-write degrades by up to 95% when the pattern performs I/O accesses. This is clearly visible, as the file-close rate of OpenFOAM exhibits intermittent behavior under interference. Comparing OpenFOAM to our open-write-close micro-benchmark, we see that it suffers in a similar way when exposed to aggregate-write, that is, its performance degrades significantly.

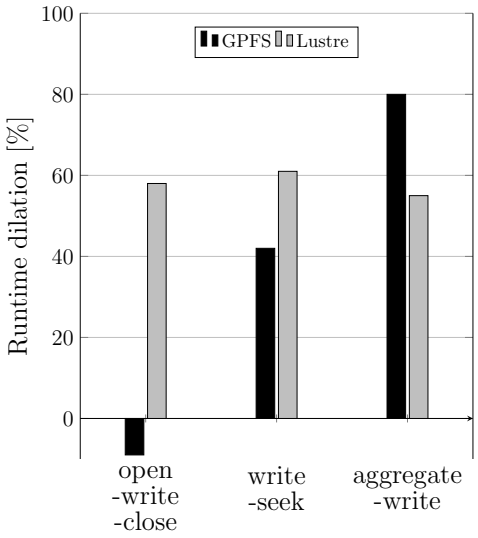


Figure 12. Throughput degradation of MadBench2 when run against different patterns

3.4.2. MadBench2

MADbench2 is derived from MADCAP cosmic microwave background radiation analysis software. MADbench2 performs dense-linear-algebra calculation using ScaLAPACK [13]. It has very large memory demands and its required matrices generally do not fit in the memory. As a result, the calculated matrices get recorded to a disk and re-read when required. This means that MadBench2 performs complex I/O operations in four phases. For our experiments, as the scope of our study is write-write contentions, we concentrate on the first phase, which has only writes and seeks. The other phases are either reads or a mixture of reads and writes. We henceforth use MadBench2 to refer to the build with the first phase only.

For the experiments, we setup MadBench2 to use POSIX I/O in the one-file-per-process mode. To maximize performance, we used the configuration recommended by Borill et al. [34], which is: WMOD=1, NPIX=50,000, NBIN=36, NGANGS=1, SBBLOCKSIZE=1, FBBLOCKSIZE=128. Furthermore, as our focus is on file-access patterns, MadBench2 is configured to run in I/O mode. In I/O mode, MadBench2 acts as a pure I/O benchmark, replacing computation with busy-wait cycles. With this configuration, and using 256 processes, MadBench2 writes 670 GB of data, with each process, performing seeks with an offset of about 74 MB during execution. This makes the I/O behavior similar to the write-seek pattern with a chunk size of 74 MB. For this reason, we executed MadBench2 against the three patterns at a chunk size of 64 MB. The results are shown in Fig. 12.

The throughput degradation on both file systems is quite different for MadBench2. On GPFS, aggregate-write generates the most interference, reducing the throughput by about 80%. Similarly, write-seek degrades the throughput of MadBench2 by about 40%. However, in the case of open-write-close, MadBench2's runtime improves. For a chunk size of 64 MiB, the higher interference aggregate-write generates is consistent with our micro-benchmarks results. On Lustre, all the patterns generate similar interference levels, with aggregate-write degrading the throughput of MadBench2 slightly less compared to others. The reason is that, for write-seek, the interference trend already reverses at a chunk size of 64 MiB, as shown in Fig. 6b. Overall, the passive interference behavior of MadBench2 resembles that of our write-seek micro-benchmark.

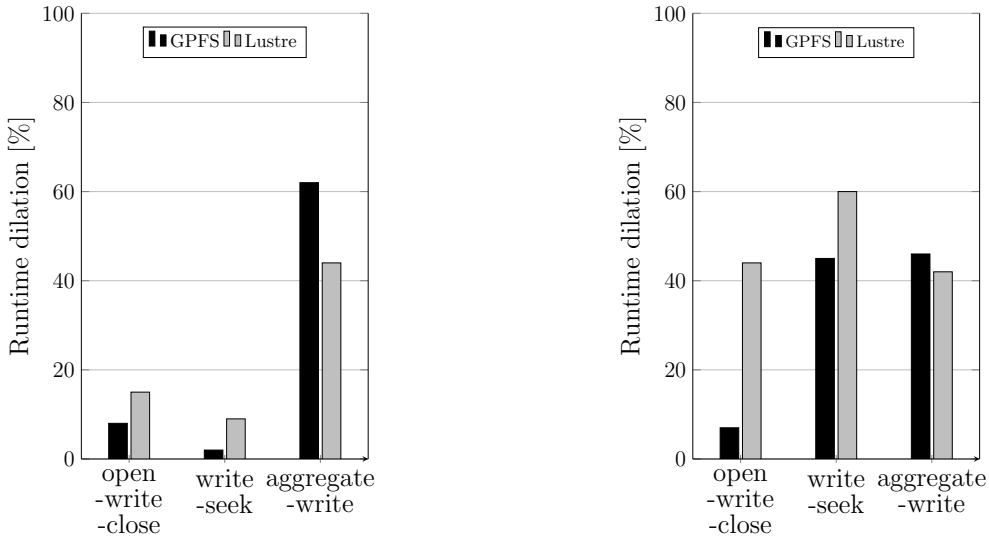


Figure 13. Throughput degradation of HACCIO when run against different patterns

3.4.3. HACCIO

HACCIO is an I/O benchmark code derived from a cosmology software framework called HACC (Hardware Accelerated Cosmology Code). HACC simulates the formation of collisionless fluids under the influence of gravity using N-body techniques. HACC has very high I/O demands, where a small simulation can write terabytes of data [7].

HACCIO writes large checkpoint files during its execution. It creates one file per process, and incrementally writes data to it. During checkpointing, the files are written, read back, and verified. As our work concentrates on write-write contention, we removed the read-back and verification part in our experiments. HACCIO can use different I/O modes during execution, including POSIX I/O, MPI with one-file-per-process or MPI with one or more shared-file.

In our experiments, we ran HACCIO with 256 processes and with POSIX I/O. During execution, each process wrote 3.6 GiB of data to its file, in chunks of 381 MiB. The I/O behavior can be equated to the aggregate-write pattern with a chunk size of 381 MiB. We ran HACCIO against the three patterns with chunk sizes of 1 MiB and 256 MiB, respectively. The results are shown in Fig. 13.

With 1 MiB, on both GPFS and Lustre, open-write-close and write-seek degrade HACCIO's performance to a smaller degree than aggregate-write. This trend is consistent with our micro-benchmark results. In the case of aggregate-write, the degradation that HACCIO suffers on GPFS is slightly higher than the one on Lustre (60% vs. 40%). In our micro-benchmark experiments for 1 MiB and 16 MiB, we also saw aggregate-write suffering a degradation of around 40% on Lustre and one between 50% and 60% on GPFS. With 256 MiB on Lustre, the degradation caused by write-seek grows to about 60%, while open-write-close and aggregate-write cause around 50% degradation. This is again similar to what has been observed with micro-benchmarks. Write-seek dominates aggregate-write at large chunk sizes. On GPFS, open-write-close degrades HACCIO by less than 10%. On the other hand, write-seek and aggregate-write cause about 50% degradation of the HACCIO write throughput. Overall, the trend is similar to our micro-benchmark-based observations.

3.4.4. Application vs. Application

With our knowledge of how isolated access patterns interfere with realistic applications, we also investigated the interference between realistic applications, as it can occur in a live production system. For this purpose, we ran OpenFOAM, MADBench2, and HACCIO first against themselves and later against each other, always using 256 processes per application. The results are shown in Fig. 14. In the figure, the x-axis shows the probe applications whose runtime dilation is reported. For each probe application, we show a separate bar for each signal application that is causing a degraded performance. In these figures, each application represents an access pattern; however, each one of them has a different chunk size and access frequency. Therefore, the interpretation of our results requires consideration of the pattern type, chunk size, and access frequency.

On GPFS, HACCIO generates the biggest interference of all, with OpenFOAM being degraded by more than 90% and MadBench2 by more than 60%. The values are similar to open-write-close against aggregate-write at a chunk size of 1 MiB and write-seek against aggregate-write at 256 MiB. MadBench2 degrades OpenFOAM by more than 80% and HACCIO by more than 10%. Here, MadBench2's behavior diverges from write-seek, with high degradation for OpenFOAM and low degradation for HACCIO. A possible explanation for OpenFOAM against MadBench2 can be the large chunk-size difference, while for HACCIO it can be low access frequency, as was observed for periodic probe signals in Fig. 7a. OpenFOAM against the other two applications generates a comparatively small throughput degradation. This is similar to our observation of open-write-close at small chunk sizes.

On Lustre, HACCIO degrades OpenFOAM by about 60%, while being degraded itself by less than 10%, similar to what was observed with micro-benchmarks. HACCIO degrades MadBench2 by about 55%, while being degraded itself by about 40%. This is again similar to micro-benchmark results, where for chunk sizes greater than 16 MiB, write-seek dominates over aggregate-write. Looking at MadBench2 against OpenFOAM, we see that OpenFOAM's runtime is dilated by about 70%. This is because of the large chunk-size difference between OpenFOAM and MadBench2.

Considering the different access patterns, write chunk sizes, and access frequencies, the overall results are in line with our observations of synthetic micro-benchmarks.

4. Related Work

Several earlier studies identified typical I/O access patterns of HPC applications. Miller et al. found I/O to be bursty and cyclic [23]. They also distinguished three access patterns, namely required I/O, checkpointing, and data staging as the most common I/O types. These patterns roughly correspond to our aggregate-write, open-write-close, and write-seek patterns, respectively. However, they were studied to optimize I/O from a single-application perspective, while we look at their interference potential when executed concurrently.

Byna et al. classified file access patterns to generate I/O-access signatures of applications [12]. These signatures were then used to improve data prefetching. Shan et al. created a parameterized I/O benchmark called IOR that can mimic the file access pattern of realistic applications [14]. Lofstead et al. found six common read patterns in the analysis part of simulation software [37]. The read patterns were used to compare end-to-end performance of logically contiguous and log-based files. Congiu et al. manually analyzed the I/O behavior of

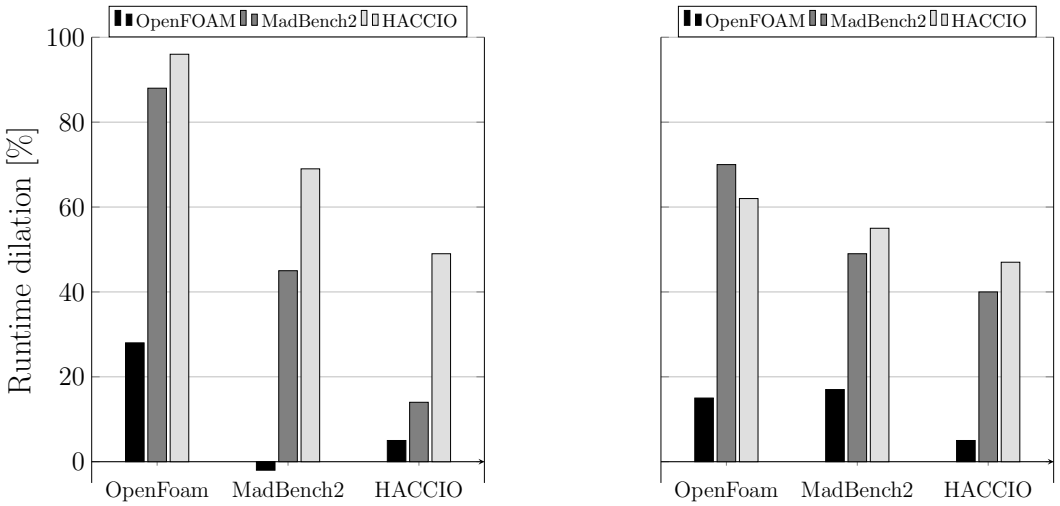


Figure 14. Throughput degradation when the applications are run against each other

applications to identify their patterns [20]. A framework, transparent to the application, then translated the knowledge of these patterns into hints to the parallel file system. Lu et al. analyzed patterns in collective I/O and found that the access pattern of a process can be lost after aggregation, negatively impacting cache performance [1]. To mitigate this effect, they proposed a cache-management policy aware of collective I/O. In our work, we evaluate the interference potential of I/O access patterns in concurrent execution.

Similarly, as part of the SIO initiative, Smirni et al. classified I/O accesses according to their spatial and temporal patterns [18, 25]. Nieuwejaar et al. classified file accesses with respect to access size, file size, access frequency, sequentiality, etc. in the CHARISMA project [39]. These studies are orthogonal to our work and part of the broader field of file-access characterization. More recent work on the topic includes characterizing read access patterns of applications with the goal of optimizing reads for subsequent data analysis and visualization [37]. Liu et al. analyzed server-side logs to identify I/O intensive applications and characterize their workloads, providing recommendation to an I/O-aware scheduler [4]. Our work studies the effects of write patterns on the I/O performance of other co-scheduled applications.

I/O performance has been the subject of several studies, looking at the performance from a single application perspective [34], from the file-system perspective [26], and from the overall system perspective [2, 11]. Further studies considered I/O interference between different jobs, identifying variability [36], uncovering performance problems with statistical techniques [32], and mitigating I/O interference through application coordination and scheduling [29]. In this paper, we analyze how file writes of concurrently running jobs interfere and determine factors that influence the magnitude of interference. While the application process count is already known as one of the factors [29], we consider process count in the context of access patterns and examine the influence of further parameters such as write-chunk size and access frequency on write performance.

SIOX records I/O accesses at each level of the I/O stack, identifies access patterns, and characterize the I/O subsystem [27, 38] with the objective of pinpointing I/O bottlenecks. Our work contributes insights into write performance variation as a result of access patterns and request sizes.

Yildiz et al. studied the root cause of inter-application I/O interference in HPC storage systems by comparing the impact of different factors [19]. They found that bad flow of control in the I/O path caused interference in most cases. Whereas they looked at I/O interference from the storage perspective, this paper takes an application-centric view.

Inter-application interference in general has also been subject of several studies. Skinner et al. identified it as one of the five sources of performance variability [31]. Shah et al. established a framework for correlating application performance across job boundaries and found I/O to be highly susceptible to the overall system load [9]. Bhatele et al. observed communication performance to be strongly influenced by co-scheduled applications on Hopper, a Cray XE system [35]. Finally, Shah et al. developed a framework to estimate the impact of inter-application interference on the execution time of bulk-synchronous MPI applications [10].

Several tools have been used to profile and monitor I/O performance of applications. Carns et al. used Darshan to characterize I/O of applications at the system level [15, 16]. Uselton et al. extended IPM for their statistical study of I/O performance variation [32]. We used LWM^2 for our study because of its ability to generate synchronized, segmented profiles that allow the performance of co-scheduled applications to be precisely correlated [9].

Conclusion

In this study, we analyzed inter-application interference effects caused by the interaction between various I/O access patterns, classified by their behavior, write chunk size, access frequency, process count, and sharing mode. Specifically, we found that at small chunk sizes data-intensive applications may significantly slow down checkpointing-intensive applications, even at smaller process counts, but not vice versa. In one case, the runtime of a checkpointing-intensive application was dilated by a factor of five. But the direction of the interference is continuously reversed as the chunk size is increased.

Given the shared nature of the majority of parallel file systems, preventing I/O interference altogether is challenging. As a general strategy to reduce it, one should try to separate I/O traffic with high interference potential either in space or in time. However, in order to make such a separation successful, it is important to decide what traffic should be separated. Leveraging techniques demonstrated now with LWM^2, file systems could be extended in the future to recognize aggressive or sensitive patterns automatically, and dynamically separate them either in space or in time. For example, traffic to a specific set of files could be (re-)routed to a specific group of file servers or buffered locally to be written back at a later point in time.

In order to support the future interference-aware file-system designs, we plan to further extend LWM^2 to recognize application I/O access patterns automatically and suggest some appropriate I/O resource scheduling policies. To this end, we want to take more complicated patterns, chunk sizes, and I/O frequencies into account with the objective of building a reliable I/O performance interference model based upon quantifiable application I/O characteristics. The interference model would also pay attention to higher-level file formats such as NetCDF and HDF5. Finally, with LWM^2's global time-slice view and the ability to detect interference through correlation, we also see machine learning techniques as a promising research direction for the prediction of interference and ultimately for its avoidance.

Acknowledgements

This research was supported by JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration) Grant No. JPMJCR1303 and the G8 Research Councils Initiative on Multilateral Research, Interdisciplinary Program on Application Software towards Exascale Computing for Global Scale Issues. Additional support was provided by the German Research Foundation (DFG) through the program Performance Engineering for Scientific Software and the US Department of Energy under Grant No. DE-SC0015524. Finally, we would like to acknowledge the support of Global Scientific Information and Computing Center at Tokyo Institute of Technology for giving us access to their system.

References

1. Lu, Y., Chen, Y., Latham, R., Zhuang, Y.: Revealing applications' access pattern in collective I/O for cache management. In: Proceedings of the 28th ACM International Conference on Supercomputing, ICS, Munich, Germany, June 10-13, 2014. pp. 181–190. ACM (2014), DOI: 10.1145/2597652.2597686

2. Yu, W., Vetter, J., Oral, H.: Performance characterization and optimization of parallel I/O on the Cray XT. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Miami, FL, USA, April 14-18, 2008. pp. 1–11. IEEE Computer Society (2008), DOI: 10.1109/IPDPS.2008.4536277

3. IBM: An Introduction to GPFS Version 3.5. `http://www-03.ibm.com/systems/resources/introduction-to-gpfs-3-5.pdf` (2014), accessed: 2014-08-11

4. Liu, Y., Gunasekaran, R., Ma, X., Vazhkudai, S.S.: Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC16, Salt Lake City, UT, USA, November 13-18, 2016. pp. 819–829. IEEE Computer Society (2016), DOI: 10.1109/SC.2016.69

5. Xie, B., Chase, J., Dillow, D., Drokin, O., Klasky, S., Oral, S., Podhorszki, N.: Characterizing output bottlenecks in a supercomputer. In: Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12, Salt Lake City, UT, USA, November 10-16, 2012. pp. 8:1–8:11. IEEE Computer Society (2012), DOI: 10.1109/SC.2012.28

6. Kuo, C.S., Shah, A., Nomura, A., Matsouka, S., Wolf, F.: How file access patterns influence interference among cluster applications. In: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER, Madrid, Spain, September 22-26, 2014. pp. 1–8. IEEE Computer Society (2014), DOI: 10.1109/CLUSTER.2014.6968743

7. Hal Finkel: Cosmic Structure Probes of the Dark Universe(Porting and Tuning HACC on Mira). `https://www.alcf.anl.gov/files/darkuniverseesptechreportwrapped.pdf` (2014), accessed 2014-08-11

8. The National Institute for Computational Sciences: I/O and Lustre Usage. https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips (2014), accessed: 2014-08-11

9. Shah, A., Wolf, F., Zhumatiy, S., Voevodin, V.: Capturing inter-application interference on clusters. In: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER, Indianapolis, IN, USA, September 23-27, 2013. pp. 1–5. IEEE Computer Society (2013), DOI: 10.1109/CLUSTER.2013.6702665

10. Shah, A., Müller, M.S., Wolf, F.: Estimating the impact of external interference on application performance. In: Proceedings of the Euro-Par 2018: Parallel Processing, Turin, Italy, August 27-31, 2018. Lecture Notes in Computer Science, vol. 11014, pp. 46–58. Springer, Cham (2018), DOI: 10.1007/978-3-319-96983-1_4

11. Lang, S., Carns, P., Latham, R., Ross, R., Harms, K., Allcock, W.: I/O performance challenges at leadership scale. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'09, New York, NY, USA, November 14-20, 2009. pp. 40:1–40:12. IEEE Computer Society (2009), DOI: 10.1145/1654059.1654100

12. Byna, S., Chen, Y., Sun, X.H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'08, Piscataway, NJ, USA, November 15-21, 2008. pp. 44:1–44:12. IEEE Computer Society (2008), DOI: 10.1109/SC.2008.5213604

13. Choi, J., Dongarra, J.J., Pozo, R., Walker, D.W.: ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In: Proceedings of the IEEE Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean, VA, USA, October 19-21, 1992. pp. 120–127. IEEE Computer Society (1992), DOI: 10.1109/FMPC.1992.234898

14. Shan, H., Antypas, K., Shalf, J.: Characterizing and predicting the I/O performance of hpc applications using a parameterized synthetic benchmark. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'08, Austin, TX, USA, November 15-21, 2008. pp. 42:1–42:12. IEEE Computer Society (2008), DOI: 10.1109/SC.2008.5222721

15. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. In: Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST, Denver, CO, USA, May 23-27, 2011. vol. 1, pp. 1–14. IEEE Computer Society (2011), DOI: 10.1109/MSST.2011.5937212

16. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of petascale I/O workloads. In: Proceedings of the IEEE International Conference on Cluster Computing and Workshops, CLUSTER, New Orleans, LA, USA, August 31-September 04, 2009. pp. 1–10. IEEE Computer Society (2009), DOI: 10.1109/CLUSTR.2009.5289150

17. Jasak, H., Jemcov, A., Tukovic, Z.: OpenFOAM: a C++ library for complex physics simulations. In: Proceedings of the International workshop on coupled methods in numerical dynamics, IUC, Dubrovnik, Croatia, September 19-21, 2007. pp. 1–20 (2007)

18. Smirni, E., Aydt, R., Chien, A., Reed, D.: I/O requirements of scientific applications: an evolutionary view. In: Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing (HPDC'96), Syracuse, NY, USA, August 06-09, 1996. pp. 49–59. IEEE Computer Society (1996), DOI: 10.1109/HPDC.1996.546173

19. Yildiz, O., Dorier, M., Ibrahim, S., Ross, R., Antoniu, G.: On the root causes of cross-application I/O interference in HPC storage systems. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS, Chicago, IL, USA, May 23-27, 2016. pp. 750–759. IEEE Computer Society (2016), DOI: 10.1109/IPDPS.2016.50

20. Congiu, G., Grawinkel, M., Padua, F., Morse, J., Süß, T., Brinkmann, A.: Mercury: A transparent guided I/O framework for high performance I/O stacks. In: Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP'17, St. Petersburg, Russia, March 06-08, 2017. pp. 46–53. IEEE Computer Society (2017), DOI: 10.1109/PDP.2017.83

21. Kunkel, J., Ludwig, T.: Performance evaluation of the PVFS2 architecture. In: Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, PDP'07, Washington, DC, USA, February 7-9, 2007. pp. 509–516. IEEE Computer Society (2007), DOI: 10.1109/PDP.2007.65

22. Dennis, J.M., Edwards, J., Loy, R., Jacob, R., Mirin, A.A., Craig, A.P., Vertenstein, M.: An application-level parallel I/O library for earth system models. International Journal of High Performance Computing Applications 26(1), 43–53 (2012), DOI: 10.1177/1094342011428143

23. Miller, E.L., Katz, R.H.: Input/output behavior of supercomputing applications. In: Proceedings of the 1991 ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis, SC'91, Albuquerque, NM, USA, November 18-22, 1991. pp. 567–576. IEEE Computer Society (1991), DOI: 10.1145/125826.126133

24. Dillow, D.A., Fuller, D., Wang, F., Oral, H.S., Zhang, Z., Hill, J.J., Shipman, G.M.: Lessons learned in deploying the worlds largest scale Lustre file system. Tech. rep., Oak Ridge National Laboratory (ORNL); Center for Computational Sciences (2010)

25. Smirni, E., Reed, D.: Workload characterization of input/output intensive parallel applications. In: Marie, R., Plateau, B., Calzarossa, M., Rubino, G. (eds.) Computer Performance Evaluation Modelling Techniques and Tools, Lecture Notes in Computer Science, vol. 1245, pp. 169–180. Springer Berlin Heidelberg (1997), DOI: 10.1007/BFb0022205

26. Kunkel, J., Ludwig, T.: Bottleneck detection in parallel file systems with trace-based performance monitoring. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008 – Parallel Processing, Lecture Notes in Computer Science, vol. 5168, pp. 212–221. Springer Berlin Heidelberg (2008), DOI: 10.1007/978-3-540-85451-7_23

27. Zimmer, M., Kunkel, J., Ludwig, T.: Towards self-optimization in HPC I/O. In: Kunkel, J., Ludwig, T., Meuer, H. (eds.) Supercomputing, Lecture Notes in Computer Science, vol. 7905, pp. 422–434. Springer Berlin Heidelberg (2013), DOI: 10.1007/978-3-642-38750-0_32

28. Carter, J., Borrill, J., Oliker, L.: Performance characteristics of a cosmology package on leading HPC architectures. In: Bougé, L., Prasanna, V. (eds.) High Performance Computing - HiPC 2004, Lecture Notes in Computer Science, vol. 3296, pp. 176–188. Springer Berlin Heidelberg (2005), DOI: 10.1007/978-3-540-30474-6_23

29. Dorier, M., Antoniu, G., Ross, R., Kimpe, D., Ibrahim, S.: Calciom: Mitigating I/O interference in hpc systems through cross-application coordination. In: Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS, Phoenix, AZ, USA, May 19-23, 2014. IEEE Computer Society (2014), DOI: 10.1109/IPDPS.2014.27

30. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series 131(1), 273 (2000), DOI: 10.1086/317361

31. Skinner, D., Kramer, W.: Understanding the causes of performance variability in HPC workloads. In: Proceedings of the IEEE International Workload Characterization Symposium, Austin, TX, USA, October 06-08, 2005. pp. 137–149. IEEE Computer Society (2005), DOI: 10.1109/IISWC.2005.1526010

32. Uselton, A., Howison, M., Wright, N., Skinner, D., Keen, N., Shalf, J., Karavanic, K., Oliker, L.: Parallel I/O performance: From events to ensembles. In: Proceedings of the IEEE International Symposium on Parallel Distributed Processing, IPDPS, Atlanta, GA, USA, April 19-23, 2010. pp. 1–11. IEEE Computer Society (2010), DOI: 10.1109/IPDPS.2010.5470424

33. Hurrell, J.W., Holland, M., Gent, P., Ghan, S., Kay, J.E., Kushner, P., Lamarque, J.F., Large, W., Lawrence, D., Lindsay, K., et al.: The Community Earth System Model: A Framework for Collaborative Research. Bulletin of the American Meteorological Society 94(9), 1339–1360 (2013), DOI: 10.1175/BAMS-D-12-00121.1

34. Borrill, J., Oliker, L., Shalf, J., Shan, H.: Investigation of leading HPC I/O performance using a scientific-application derived benchmark. In: Proceedings of the ACM/IEEE Conference on High Performance Computing Networking, Storage and Analysis SC'07, Reno, NV, USA, November 10-16, 2007. pp. 1–12. IEEE Computer Society (2007), DOI: 10.1145/1362622.1362636

35. Bhatele, A., Mohror, K., Langer, S.H., Isaacs, K.E.: There goes the neighborhood: performance degradation due to nearby jobs. In: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '13, Denver, CO, USA, November 17-22, 2013. IEEE Computer Society (2013), DOI: 10.1145/2503210.2503247

36. Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., Wolf, M.: Managing variability in the IO performance of petascale storage systems. In: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10), New Orleans, LA, USA, November 13-19, 2010. pp. 1–12. IEEE Computer Society (2010), DOI: 10.1109/SC.2010.32

37. Lofstead, J., Polte, M., Gibson, G., Klasky, S., Schwan, K., Oldfield, R., Wolf, M., Liu, Q.: Six degrees of scientific data: Reading patterns for extreme scale science IO. In:

Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC'11, San Jose, California, USA, June 08-11, 2011. pp. 49–60. ACM (2011), DOI: 10.1145/1996130.1996139

38. Wiedemann, M., Kunkel, J., Zimmer, M., Ludwig, T., Resch, M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W., Kluge, M., Mickler, H.: Towards I/O analysis of HPC systems and a generic architecture to collect access patterns. Computer Science - Research and Development 28(2-3), 241–251 (2013), DOI: 10.1007/s00450-012-0221-5

39. Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C., Best, M.: File-access characteristics of parallel scientific workloads. IEEE Transactions on Parallel and Distributed Systems 7(10), 1075–1089 (October 1996), DOI: 10.1109/71.539739

40. Laboratory, E.O.L.B.N.: Global cloud resolving model simulations, ernest orlando lawrence berkeley national laboratory. `http://vis.lbl.gov/Vignettes/Incite19` (2014), accessed: 2014-08-11

DOI: 10.14529/jsfi190204

Investigating the Dirac Operator Evaluation with FPGAs

Grzegorz Korcyl[1], *Piotr Korcyl*[2,3]

In recent years, computational capacity of single Field Programmable Gate Array (FPGA) devices as well as their versatility have increased significantly. Adding to that fact, the High Level Synthesis frameworks allowing to program such processors in a high-level language like C++, makes modern FPGA devices a serious candidate as building blocks of a general-purpose High Performance Computing solution. In this contribution we describe benchmarks which we performed using a kernel from the Lattice QCD code, a highly compute-demanding HPC academic code for elementary particle simulations on the newest device from Xilinx, the U250 accelerator card. We describe the architecture of our solution and benchmark its performance on a single FPGA device running in two modes: using either external or embedded memory. We discuss both approaches in detail and provide assessment for the necessary memory throughput and the minimal amount of resources needed to deliver optimal performance depending on the available hardware. Our considerations can be used as guidelines for estimating the performance of some larger, many-node systems.

Keywords: high performance computing, FPGA, lattice QCD, Dirac operator evaluation.

Introduction

Quantum Chromodynamics is the theory describing the interactions of quarks and gluons, explaining why the latter form bound states such as protons and neutrons. One of the characteristic features of this theory is that quarks and gluons form a strongly coupled system in the low energy regime. As a consequence, it is difficult to extract predictions for the properties of such a system from First Principles of Physics. Up to now, the only available computational tool allowing for such calculations are numerical simulations (Monte Carlo simulations) of a discretized version of the theory, called Lattice Quantum Chromodynamics (LQCD). Traditionally, physicists working in the field of LQCD searched for the most performant, vector machines consisting of a large number of compute nodes, and have designed many new HPC solutions: QCDOC [5], APE [1], QPACE [3], just to name a few. Currently, GPU and ARM processors are considered for the next generation of supercomputing machines and it is an open question whether FPGA devices could be used as an alternative.

In the discretized version of Quantum Chromodynamics the basic degrees of freedom are associated to each point of a four-dimensional grid representing a finite volume of four-dimensional space-time. Sizes of such volumes vary from $V = 10^6$ up to $V = 10^8$ points. The most compute-intensive part of any such simulation is the inversion of the Dirac matrix, which is of size $(24V) \times (24V)$. The matrix has a sparse structure because it describes the nearest-neighbour interactions. The Dirac matrix $D(n,m)^{AB}_{\alpha\beta}$ acting on the vector $\psi(n)$ can be written down as follows [9]

[1]Department of Information Technologies, Faculty of Physics, Astronomy and Applied Computer Science, Jagiellonian University, Kraków, Poland
[2]Institut für Theoretische Physik, Universität Regensburg, Regensburg, Germany
[3]M. Smoluchowski Institute of Physics, Jagiellonian University, Kraków Poland

$$D(n,m)^{AB}_{\alpha\beta}\psi^B_\beta(m) = (m_q+4)\psi^A_\alpha(n) + \\ + \frac{1}{2}\sum_{\mu=0}^{3}\Big[U^{AB}_\mu(n)P^{-\mu}_{\alpha\beta}\psi^B_\beta(n+\hat{\mu}) + U^{\dagger,AB}(n-\hat{\mu})P^{+\mu}_{\alpha\beta}\psi^B_\beta(n-\hat{\mu})\Big]. \quad (1)$$

The most elementary computational block is the evaluation of the single stencil, i.e. evaluation of the right hand side of (1) for a given value of index n. Note that the coefficients of $D(n,m)^{AB}_{\alpha\beta}$ matrix differ for each m, i.e. U complex-valued 3×3 matrices and ψ complex-valued 3-element vectors depend on position m. Therefore, each stencil involves loading of eight $U(n)$ matrices and nine spinor fields from the neighboring lattice sites, which in total corresponds to 360 input words. In case of double precision, this amounts to 2880 input bytes. One can exploit the structure of $SU(3)$ matrices and parametrize them in terms of 10 input words each, instead of 18 in the naive formulation (9 real and 9 imaginary entries). We return to this point in Section 2.2. $U\times\psi$ matrix-vector multiplications require 1464 floating point operations for complex additions and multiplications. $P^{\pm}$ are real-valued 4×4 constant matrices, m_q is a real parameter corresponding to the quark mass, μ labels directions in the four-dimensional space-time. Repeated indices are summed within the ranges: $\alpha,\beta = 1,\ldots,4$, $A,B = 1,2,3$. For unexplained notation, please see [9] or [7]. One of the simplest algorithms allowing to invert such a matrix is an iterative conjugate gradient algorithm. The relevance of this algorithm is demonstrated by the fact that HPCG benchmark has been introduced since November 2017 as a new ranking of supercomputers published by TOP500 organization. Such benchmark differs from the traditionally used Linpack benchmark where the employed matrix was dense. The argument behind the HPCG benchmark is that sparse matrix computations in many cases are more representative of the variety of HPC applications which run on a supercomputer. Indeed, the iterative solver of the type of conjugate gradient is, for instance, at the heart of Monte Carlo simulation of QCD.

The rest of this article is organized as follows. In the next section we specify the details of the implemented algorithm as well as summarize the description of the kernel which is being hardware-accelerated. Subsequently in the following Section, we propose two implementations on the FPGA devices which differ by the location where the main data is stored, either these are registers in the programmable logic, or an external DDR memory bank attached to the programmable logic. In Section 3, we compare and discuss the achieved performances using both approaches. Eventually, we conclude and point out future research directions.

1. Kernel Description

In this work, we consider an improved version of the conjugate gradient algorithm which allows us to test different floating and fixed point precisions without a deterioration of the ultimate solution. Similar considerations for GPU were presented in [4]. The algorithm intertwines iterations in low and high precision, working mainly in low precision and correcting a possible systematic error by a high precision iteration. Our algorithm follows the one suggested in [8] and is shown in Algorithm 1. We provide an exact form of the mixed precision conjugate gradient algorithm implemented in this work to show which parts have been hardware accelerated and what is the interplay between parts of the algorithm requiring implementations in different precision. In both cases, the most time consuming part is matrix multiplications in lines 2, 14 and 24 of Algorithm 1.

Algorithm 1 Residual Guided CG algorithm

$\psi^{\text{high}} \leftarrow \psi_0^{\text{high}}$
$r_0^{\text{high}} \leftarrow \eta^{\text{high}} - \left(D^\dagger D\right)^{\text{high}} \psi^{\text{high}}$
$s_0^{\text{high}} \leftarrow ||r_0^{\text{high}}||$
$r_0 \leftarrow \frac{r_0^{\text{high}}}{s_0^{\text{high}}}$
$l \leftarrow 0$
while $s^{\text{high}} \geq r_{min}^{\text{high}}$ **do**
  $n \leftarrow 0$
  $\psi_0 \leftarrow 0$
  $r_0 \leftarrow \frac{r_{l+1}^{\text{high}}}{s_{l+1}^{\text{high}}}$
  $p_0 \leftarrow p_k - \left(r_0 \cdot p_k\right) r_0$
  $\alpha_0 \leftarrow 0$
  $\beta_0 \leftarrow \frac{s_{l+1}^{\text{high}}}{s_l^{\text{high}} \rho_k}$
  while $n < k$ **do**
    $q_n \leftarrow D^\dagger D p_n$
    $\alpha_n \leftarrow \frac{\rho_n}{p_n \cdot q_n}$
    $\psi_{n+1} \leftarrow \psi_n + \alpha_n p_n$
    $r_{n+1} \leftarrow r_n - \alpha_n q_n$
    $\rho_{n+1} \leftarrow r_{n+1} \cdot r_{n+1}$
    $\beta_n \leftarrow \frac{\rho_{n+1}}{\rho_n}$
    $p_{n+1} \leftarrow r_{n+1} + \beta_n p_n$
    $n \leftarrow n + 1$
  end while
  $\psi_{l+1}^{\text{high}} \leftarrow \psi_l^{\text{high}} + s_l^{\text{high}} \left(\psi_k + \alpha_k p_k\right)$
  $r_{l+1}^{\text{high}} \leftarrow b^{\text{high}} - \left(D^\dagger D\right)^{\text{high}} \psi_{l+1}^{\text{high}}$
  $s_{l+1}^{\text{high}} \leftarrow ||r_{l+1}^{\text{high}}||$
  $l \leftarrow l + 1$
end while

We would like to hardware accelerate them and briefly summarize the FPGA implementation of these kernel functions. We follow what was presented in [7]. In particular that Reference contains a description of C++ data structures used for the implementation as well as relevant details of the memory allocation which allows for a fully pipelined execution of the kernel. Fragments of C++ and HLS directive codes are provided and discussed in that Reference.

For both high and low precisions of the kernel, implementation is similar: a single function involves a loop over a subvolume and an evaluation of the stencil for each site of the lattice. Evaluation of a single stencil is fully parallelized as far as data dependencies allow, this and all stencils are pipelined.

All operations involved in the estimation of a single stencil are graphically shown in Fig. 1. The evaluation naturally splits into 4 stages. The clock cycles provide an estimate of the amount of parallelization and correspond to the number of clock cycles required to finish the computation at a given stage in double precision. In the first stage, all the necessary data is copied from the BRAM memory blocks to local registers which only requires one clock cycle. In stage 2, linear combinations of input data, 8 additions and 8 subtractions of vector type are evaluated. They are

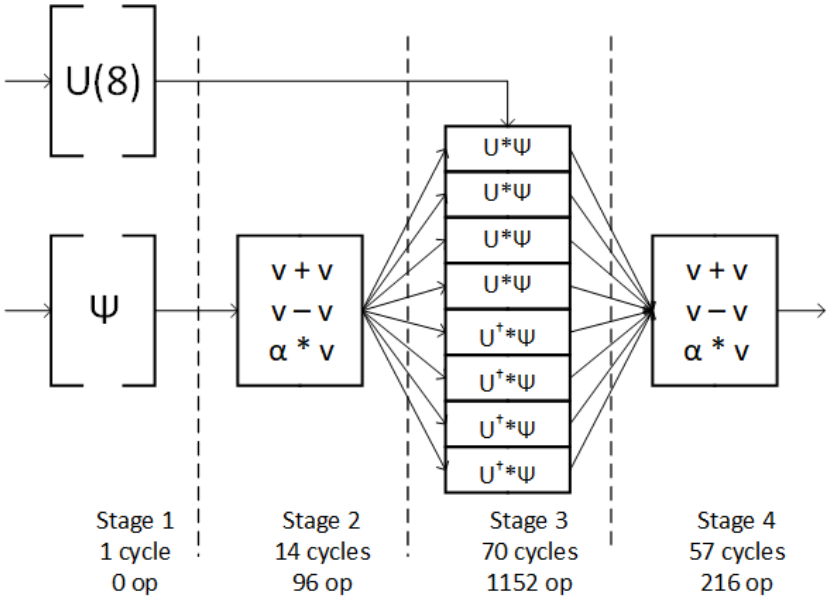


Figure 1. Computation sequence of the stencil solver

all performed in parallel, taking 14 clock cycles, which corresponds to a single addition of double numbers in programmable logic. The most compute-intensive stage 3 involves $SU(3)$ matrix by vector multiplications. In total, 1152 operations are performed. Complete parallelization allows to execute them in a 5-layer operation cascade, taking in total $5 * 14 = 70$ cycles. Finally, at stage 4, all contributions are added up to the final result. Because of the dependencies between consecutive partial results, a 4-layer operation cascade gets created, which in total takes $57 = (4 * 14) + 1$ clock cycles, 4 additions plus one data copy. Overall, the kernel requires 142 clock cycles and a total of 1464 basic operations to compute the final result since the reception of the input data. The kernel is fully pipelined: i.e. it can accept new input data at each clock cycle and produce results with latency of 142 cycles.

2. Two Approaches

There are two approaches one can follow in order to provide required data to the kernel. One can divide the entire problem into small parts so that the entire set of data for a single part fits into the BRAM memory of the device. Alternatively, one can store the entire set of data in the DDR die attached to the programmable logic and stream the data through the link. We discuss performances of the both solutions below.

2.1. A Smaller Lattice Stored in BRAM Memory

This is the approach we followed in [7]. We showed that lattices up to the size of 12×8^3 data points in each direction in double precision can fit into the internal memory of the programmable logic of the FPGA devices available currently on the market. In Fig. 2, we show the required number of URAM blocks for a given size of the lattice for single and double precision. As one can in that figure, the storage requirements are not linear because it is crucial to store data in PL in as many separate PL local registers blocks as possible in order to allow the compiler to take advantage of the natural parallelism of FPGA devices. This is due to the fact that in a single PL clock cycle only one memory element can be read from the BRAM block. In the

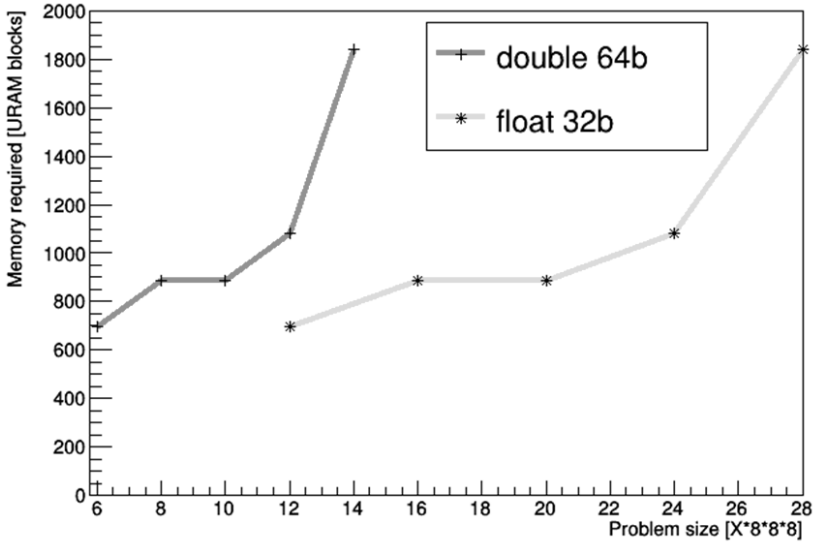


Figure 2. Memory usage as a function of the initiation interval

computation of a single stencil, one needs eight different U matrices and we insist that they are stored separately. Although this requires duplicating the amount of stored data, the matrices $U(n)$ and $U^\dagger(n)$ are stored separately, the gain is considerable. The HLS directives ensuring such memory allocation were described in [7]. Thanks to that, the stencil evaluation can be fully pipelined, i.e. the hardware block can accept new input data at each clock cycle. The resulting performance simulated in software is 812 GFLOPs for single precision and 406 GFLOPs for double precision with the PL running at 300 MHz.

2.2. A Larger Lattice Streamed from the DDR Memory

The way to operate on larger data sets is to keep the data in the DDR die attached to the programmable logic and process the data in a streaming mode. This was investigated on the Maxeler system in [6]. The U matrices and ψ spinors are prepared beforehand into sets corresponding to consecutive stencils and are streamed continuously from the DDR into the logic. The limitation of this solution is the throughput of the memory link between the DDR and the logic. Using SDAccel and an openCL implementation of the CG algorithm, we verified that one can send 256B for the Xilinx U250 device from the DDR memory to the PL part per clock cycle, working at the frequency of 300 MHz. Four channels are available aggregating to 77 GBps throughput. In order to decrease the amount of data transferred, we change the representation of U matrices, and following [2], we use a 10 parameter parametrization. We trade two more parameters and avoid computing trigonometric functions in the programmable logic. The reduced set of data translates to an initiation interval of 5 and 9 clock cycles for the compute kernel for single and double precision respectively, i.e the programmable logic has to wait 5/9 clock cycles to gather enough data to start a new computation. The performance in that case would approximately be equal to 86 and 46 GFLOPs respectively, which is comparable to the one quoted in [6] on the Maxeler system. However, if we also count the additional operations needed to recover U matrices from their reduced form, the achieved sustained performance reaches 194 GFLOPs for single precision. In Fig. 3, we show how the required throughput depends on the initiation interval. The calculation assumes the reduced form of U matrices. The smaller the initiation interval is, the shorter is the time in which the data has to be transferred.

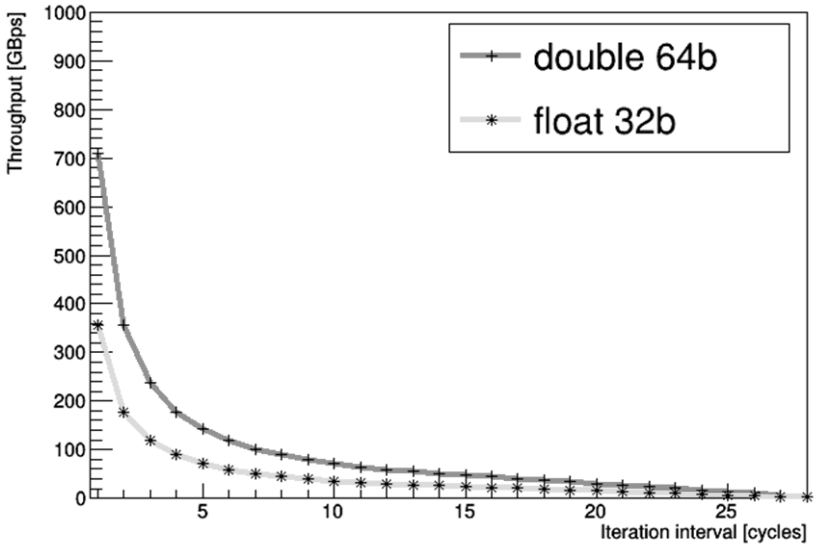


Figure 3. Transmission rates as a function of the initiation interval

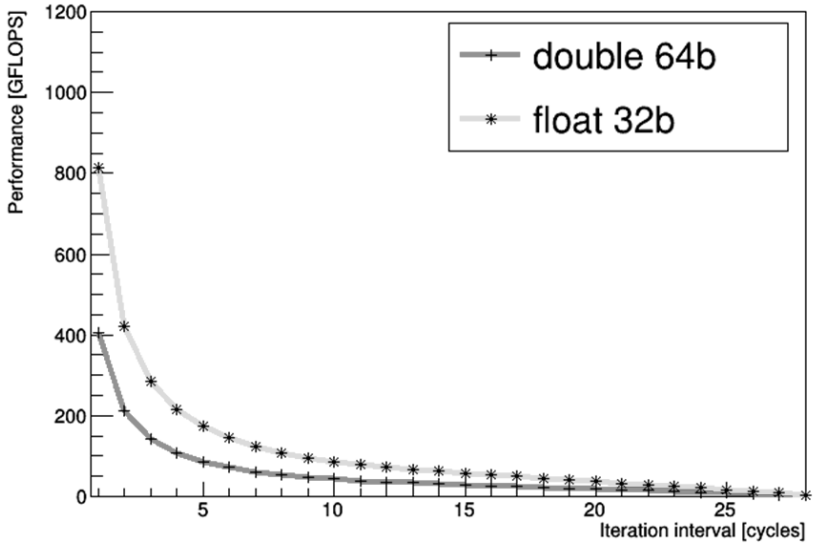


Figure 4. Performance as a function of the initiation interval

We show the throughput estimates for single and double floating point precision. Knowing the throughput between the DDR and the programmable logic on a given device, one can easily read the corresponding minimal initiation interval and henceforth the resulting performance, which is shown in Fig. 4 for both the single and double precision case.

Finally, in Fig. 5 we show how the hardware resource consumption depends on the initiation interval for single and double floating point precision but also for a more FPGA friendly 32 bit fixed point data format. In this streaming scenario one can relax the initiation interval of one clock cycle imposed in the first approach. The memory throughput being the bottleneck, one can implement the kernel with a lower initiation interval because in any case several clock cycles are needed to collect all the necessary data for a single stencil computation. The figure shows an indicative percentage of all available resources counting together all DSP, LUTs and BRAM blocks. We see that in the described case where the memory throughput imposes an initiation interval of 5 clock cycles the compute kernel uses only 20% of the available resources for double precision.

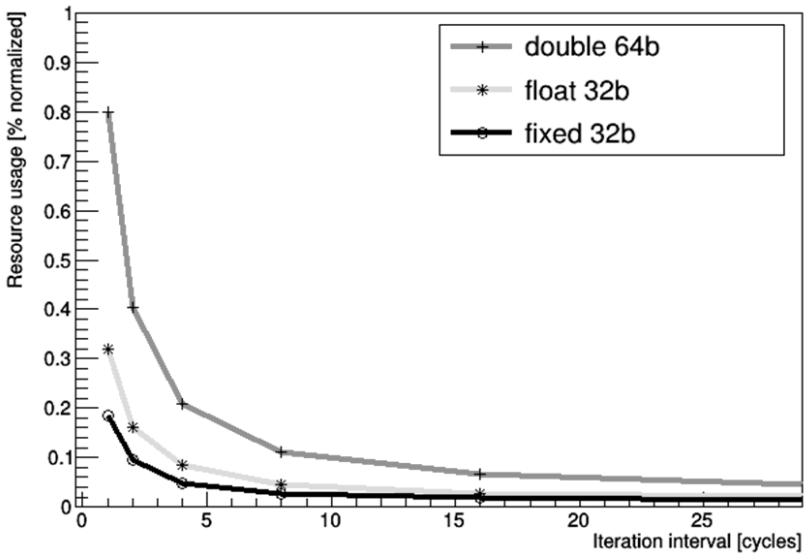


Figure 5. Resources consumption as a function of the initiation interval

3. Discussion

The presented results allow understanding various constraints limiting performance of the investigated kernel on FPGA devices. Starting from the embedded memory scenario, the practical problems that are being analyzed are larger by a factor of the order of 4096. One would probably use that amount of FPGA devices running in parallel and exchanging boundary data directly from and to the programmable logic through the embedded transceivers. On the other hand, in principle, the entire set of data could be stored in the DDR in the external memory scenario. However, the wall clock time to the solution on a single FPGA device would be impractically long. In that case, one would also resort to a many-node system where the computations could be speed up by running them in parallel. In principle, neither of the two scenarios is obviously superior. The number of required nodes can be different in both solutions and the details would depend essentially on the memory throughput of the FPGA device used. With the numbers provided above, such estimations can be put on a solid ground.

Conclusions

In this contribution, we discussed the applicability of FPGA devices to High Performance Computing solutions. We used the academic code for Monte Carlo simulations of Quantum Chromodynamics as a benchmark. In traditional computer architectures, this code is memory-bound due to the unfavorable ratio of the amount of data to be loaded to the amount of floating point operations to be executed be the most elementary kernel function. On the available programmable logic hardware, the problem turns out to be memory bound in the scenario where data is streamed from the DDR die, which will be considerably improved with the arrival of Xilinx Alveo U280 cards with a 480 GB/s memory bandwidth between DDR and programmable logic. In the scenario where data is stored in the embedded memory, the problem's limitation is the available size of the internal memory. Both cases seem to be scalable and thus offer a viable proposal for a larger scale infrastructure.

Acknowledgments

This work was in part supported by Deutsche Forschungsgemeinschaft under Grant No. SFB/TRR 55 and by the polish NCN grant No. UMO-2016/21/B/ ST2/01492, by the Foundation for Polish Science grant no. TEAM/2017-4/39 and by the Polish Ministry for Science and Higher Education grant no. 7150/E-338/M/2018. The project could be realized thanks to the support from Xilinx University Program and their donations. P.K. acknowledges support from the NAWA Bekker fellowship and thanks Università degli Studi di Roma Tor Vergata for hospitality during which this work was finalized.

References

1. APE collaboration: APE project in Rome. `http://apegate.roma1.infn.it`, accessed: 2019-04-19

2. Bunk, B., Sommer, R.: An 8 parameter representation of SU(3) matrices and its application for simulating lattice qcd. Computer Physics Communications 40(2), 229–232 (1986), DOI: 10.1016/0010-4655(86)90111-6

3. Baier, H., et al.: QPACE: A QCD parallel computer based on Cell processors. PoS LAT2009, 001 (2009), DOI: 10.22323/1.091.0001

4. Clark, M.A., Babich, R., Barros, K., Brower, R.C., Rebbi, C.: Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. 181, 1517–1528 (2010), DOI: 10.1016/j.cpc.2010.05.002

5. Boyle, P., Chen, D., Christ, N., Clark, M., Cohen, S., Cristian, C., Dong, Z., Gara, A., Jo, B., Jung, C., Kim, C., Levkova, L., Liao, X., Liu, G., Mawhinney, R., Ohta, S., Petrov, K., Wettig, T., Yamaguchi, A.: Hardware and software status of qcdoc. Nuclear Physics B - Proceedings Supplements 129-130, 838–843 (2004), DOI: 10.1016/S0920-5632(03)02729-4, lattice 2003

6. Janson, T., Kebschull, U.: Highly Parallel Lattice QCD Wilson Dirac Operator with FPGAs. Parallel Computing is Everywhere 32, 664–672, DOI: 10.3233/978-1-61499-843-3-664

7. Korcyl, G., Korcyl, P.: Towards Lattice Quantum Chromodynamics on FPGA devices (2018)

8. Strzodka, R., Goddeke, D.: Pipelined mixed precision algorithms on fpgas for fast and accurate pde solvers from low precision components. In: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. pp. 259–270. FCCM '06, IEEE Computer Society, Washington, DC, USA (2006), DOI: 10.1109/FCCM.2006.57

9. Gattringer, C., Lang, C.B.: Quantum chromodynamics on the lattice. Lect. Notes Phys. 788, 1–343 (2010), DOI: 10.1007/978-3-642-01850-3

DOI: 10.14529/jsfi190205

Development of a RISC-V-Conform Fused Multiply-Add Floating-Point Unit

Felix Kaiser[1]***, Stefan Kosnac***[2]***, Ulrich Brüning***[2]

Despite the fact that the open-source community around the RISC-V instruction set architecture is growing rapidly, there is still no high-speed open-source hardware implementation of the IEEE 754-2008 floating-point standard available. We designed a Fused Multiply-Add Floating-Point Unit compatible with the RISC-V ISA in SystemVerilog, which enables us to conduct detailed optimizations where necessary. The design has been verified with the industry standard simulation-based Universal Verification Methodology using the Specman e Hardware Verification Language. The most challenging part of the verification is the reference model, for which we integrated the Floating-Point Unit of an existing Intel processor using the Function Level Interface provided by Specman e. With the use of Intel's Floating-Point Unit we have a "known good" and fast reference model. The Back-End flow was done with Global Foundries' 22 nm Fully-Depleted Silicon-On-Insulator (GF22FDX) process using Cadence tools. We reached 1.8 GHz over PVT corners with a 0.8 V forward body bias, but there is still a large potential for further RTL optimization. A power analysis was conducted with stimuli generated by the verification environment and resulted in 212 mW.

Keywords: floating-point, multiply-add, risc-v, hardware-design, verification, uvm, synthesis, asic, gf22fdx, ieee754.

Introduction

The open-source RISC-V Instruction Set Architecture (ISA) has gotten great attention in the last years and continues to thrive. However, it has yet to enter the realm of High-Performance Computing (HPC). To enable high-performance processors based on RISC-V, it is crucial to provide fast hardware Floating-Point Units (FPUs). Arguably, the most considered ranking of HPC systems is the TOP500 [17]. The criterion for this list is the floating-point-performance based on the benchmark LINPACK [12]. LINPACK is a numerical library for linear algebra. Therefore, floating-point multiplication with subsequent additions need to be performed. This corresponds to the function of the so called Fused Multiply-Add (FMA) units. FMA units implement a multiplication and a consecutive addition without intermediate rounding in hardware. Consequentially, the high throughput and energy efficient FMA units count to the essentials in HPC hardware. That is the reason why we are focussing on them within this paper.

Due to the standardization of floating-point called IEEE 754-2008, modular FPUs and especially FMAs have already been on the market for decades. The need for a new RISC-V-specific implementation lies in the nature of that standard: For historic reasons not each statement it contains is unique. To keep the design of the ISA clean and the results of different implementations reproducible, RISC-V makes these decisions fixed, but different from the existing implementations [20].

The latter enables the possibility to develop a universal verification environment for FPUs. This is another challenging point we are tackling within this work. Due to the high number of possible input patterns, we applied a simulation-based approach following the industry standard Universal Verification Methodology (UVM). UVM intends to generate constraint-random stimuli

[1]EXTOLL GmbH, Mannheim, Germany
[2]Heidelberg University, Heidelberg, Germany

for the Design Under Test (DUT). The same stimuli is distributed to one or more reference models to generate the expected result that is to be compared with the DUT output. As a key component of this approach a reference model has to be "known good". So we used Intel Intrinsics to get low level access to the Intel FMA unit of the processor running the verification tasks. Since some details were not covered by this model, we decided to integrate Berkeley SoftFloat, which is a software implementation of the IEEE floating-point standard.

Besides the Register Transfer Level (RTL) further optimizations for speed and power can be done in the Semi-Custom part of the flow. For the last step of the implementation, we performed this Back-End design flow for Global Foundries' 22 nm Fully-Depleted Silicon-On-Insulator (FDSOI) process. This includes synthesis, floorplanning, placement, scan insertion, clock tree synthesis and routing. Therewith, we are able to analyze which target frequencies are reachable with and without Forward Body Bias (FBB) and estimate the expected power consumption.

This article is organized in four parts. Section 1 gives an overview of the state of the art, followed by Section 2, which presents the FMA unit architecture. In Section 3 we discuss our verification approach, and in Section 4 the synthesis results are presented. Last but not least a conclusion summarizes our work.

1. State of the Art

As the presented work consists of three major parts, namely the design, the implementation, and the verification, different explorations need to be done. For the goal of a high-performance unit, design and implementation have to go hand in hand. Hence, the exploration is split up into development, which consists of design and implementation, and verification.

First of all, it has to be mentioned that not each IEEE 754-2008-conform FPU can be compared with every other one. This comes from the fact that IEEE 754-2008 leaves some decisions to the designer [2]. The RISC-V Foundation decided to avoid differences in functionality between different RISC-V compliant FPUs by making these decisions fixed within their standard. Following that, we only take other RISC-V-conform FPUs in consideration.

1.1. Development

The arguably most known implementation of a RISC-V FPU is the HardFloat of the University of California, Berkeley (UC Berkeley) [19]. It is used within different cores, or core generators, like the System-on-Chip (SOC) generator Rocket [4] and the Out-of-Order core Berkeley Out-of-Order Machine (BOOM) [8]. The fastest Rocket Implementation is SiFives' U54 Rocket on the TSMC 28 nm HPC process with 1.5 GHz [7]. Due to its multiple usages and even Tape-Outs, HardFloat can be counted as reliable. However, when it comes to high-performance, it has disadvantages. HardFloat is developed using the high-level hardware generation language Chisel [5]. Generally, Chisel does not take the opportunity to optimize a design completely, but in case of HardFloat a descriptive approach instead of an optimized architecture is chosen. Following that, the whole optimization is done within the Back-End. This restricts the potential of optimizing at the RTL.

Another open-source RISC-V-conform FPU is from Parallel Ultra Low Power (PULP) [11]. PULP is a platform of the ETH Zürich, where a set of RISC-V cores and peripherals they developed are provided. There is also an FPU designed in SystemVerilog [16]. Unfortunately,

it does not provide double-precision. It is also missing the rounding mode roundTiesToAway, which is obligatory for RISC-V.

1.2. Verification

The common issue with verification is that not each possible input and state can be tested. The provided design does not have a complex state, but, due to the wide inputs, it still can not be verified by iterating through all possible values. Realizable approaches are formal or simulation-based verification. Although there have been formal verifications of FPUs in the last years [10, 14], the proposed design is verified simulation-based, due to the larger amount of time needed for the corner cases in the execution of the formal methods [6, 9].

Since a verification environment for an FMA unit does not have to react to the internal state, it can be verified by generating the stimuli statically. Also it can be generated offline, which both increases the performance of the tests. An example of such a generator is IBMs FPGen [3], which works using a constraint solver. Unfortunately, the actual generator, or constraint solver, is not open-source, only a set of pre-generated single-precision inputs. Another approach, which is even a part of the RISC-V ecosystem, is the so called TestFloat [15]. TestFloat is a similar approach, that makes use of the SoftFloat model. SoftFloat is a software implementation of the IEEE 754-2008. Even though TestFloat would work for our design, we are using an UVM-based approach as it enables an efficient integration into system- respectively chip-level testbenches.

2. FMA Unit Design

Currently, the FMA unit supports all four double-precision fused operations defined in the RISC-V ISA (Tab. 1) as well as add, subtract, and multiply. It supports all rounding modes required by the IEEE 754-2008 standard and additionally roundTiesToAway, which is mandatory for RISC-V. Divide and square root will be implemented in the future using a Newton-Raphson algorithm.

Table 1. Supported RISC-V floating-point instructions

Instruction	Description	Operation
FADD	Add	$A + C$
FSUB	Subtract	$A - C$
FMUL	Multiply	$A \cdot B$
FMADD	Fused Multiply-Add	$A \cdot B + C$
FMSUB	Fused Multiply-Subtract	$A \cdot B - C$
FNMSUB	Negative Fused Multiply-Subtract	$-A \cdot B + C$
FNMADD	Negative Fused Multiply-Add	$-A \cdot B - C$

Figure 1 shows the interface and architecture of the FMA unit, which is based on [18]. It comprises three 64-bit inputs `port_a`, `port_b`, and `port_c` for the operands, and the 64-bit wide output `port_res` for the result. The type of operation is determined by `op`, the rounding mode by `rm` and exceptions are signaled at the output `exception_flags`. A forward flow control (not shown here) is implemented via `valid_in` and `valid_out`. `valid_in` can also be used for clock-gating inside the FMA unit. In the following, the main components of the design are described in more detail. The Sign-/Exponent Transformation transforms the operand exponents from the

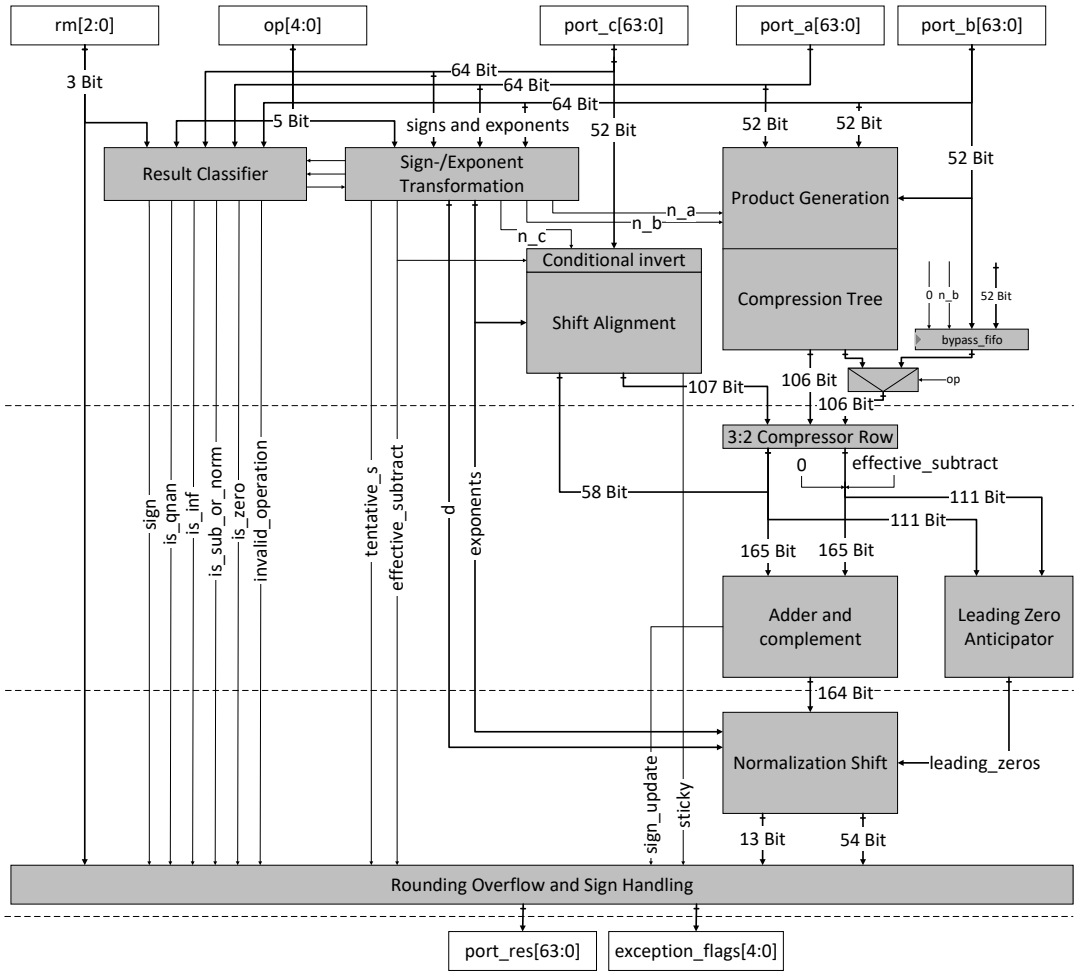


Figure 1. FMA unit architecture (the dashed lines represent the three pipeline stages)

biased representation into 2's complement and checks if the operands are normal, i.e. have no special values. Furthermore, it calculates the difference between the products' exponent (e_A+e_B) and the addends exponent (e_C). It also generates an effective subtraction bit indicating if the absolute values of $A \cdot B$ and C are added or subtracted. The Result Classifier handles operations with special values, such as qNaN, sNaN, zero, infinity, and subnormal numbers. Product Generation and Compression Tree perform the main part of the multiplication. Therefore, they take the mantissas of operands A and B and provide the product in carry-save representation. This allows to take the additional 3:2 Compressor Row to add another operand (mantissa of C) at low cost and is one of the reasons to perform FMA operations at all. For a floating-point addition, it is necessary to align the addends according to their exponent by shifting one of them relative to the other. This is done by the Shift Alignment in parallel to the compression tree. There is the case where one addend is much larger than the other, so the smaller one is completely absorbed and does not change the result. The Adder and Complement resolves the carry-save representation, as well as the following 2's complement into an 1's complement intermediate result. In parallel to the Adder, a so called Leading Zero Anticipator estimates the leading zeros of the intermediate result for the normalization. The latter is then done by the Normalization Shift. Afterwards the result is finalized by the Rounding, Overflow and Sign Handling unit, which determines if there is an overflow and performs rounding based on this information.

3. FMA Unit Verification

The verification of an FMA unit is a challenging task since a reference model is not easily developed, and not all input combinations can be tested within a reasonable time. The latter is mitigated with the simulation-based UVM, which we applied for the FMA unit. To avoid missing test cases that would show an erroneous behavior, the test cases are not predefined but instead generated constrained-random, which additionally facilitates automation. To keep track of which parts have been tested, code coverage, as well as functional coverage is used. The last important aspect is the checking. Due to its many special behaviors, the most challenging task in the verification of an FMA unit in general is to automatically generate the answer to the question whether a behavior is correct or not. Behavioral models, which are the common way to solve that issue, are usually developed by a verification engineer for the specific design. Since floating-point is standardized by IEEE 754-2008, other units can be used for this purpose.

One attempt of getting a reference for floating-point operations is to execute them in the applied language for the testbench. In last instance, such an operation maps onto the FPU within the utilized CPU. Since a higher level programming language and the instructions executed by a processor are separated by abstraction layers, this introduces a lack of controllability. For instance, the operation $D = A \cdot B + C$ may be compiled to a single multiplication followed by an addition or to an FMA operation. In our approach, we force the processor to execute the intended operation by using the programming language C and implementing the operations as intrinsics [1]. Therewith, we get the reliability of a "known good" Intel FPU.

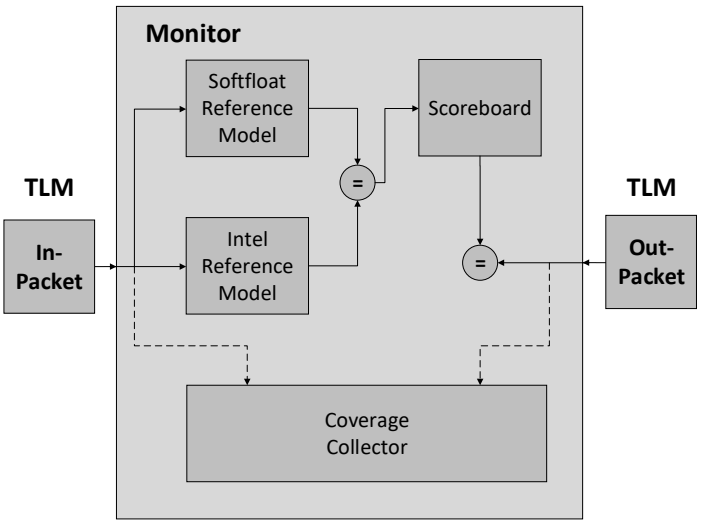


Figure 2. UVM Monitor applying the Intel Intrinsics reference model and the SoftFloat reference model

This approach alone is not sufficient as a reference model for a RISC-V FPU. As mentioned before, IEEE 754-2008 left some decisions to the implementer and Intel and the RISC-V Foundation took different choices. For these situations, we integrated another reference model, the already mentioned SoftFloat model [15]. Instead of just filling the holes using this model, we integrated both models in parallel and checked them for common cases against each other as shown in Fig. 2. Functional coverage is collected concurrently.

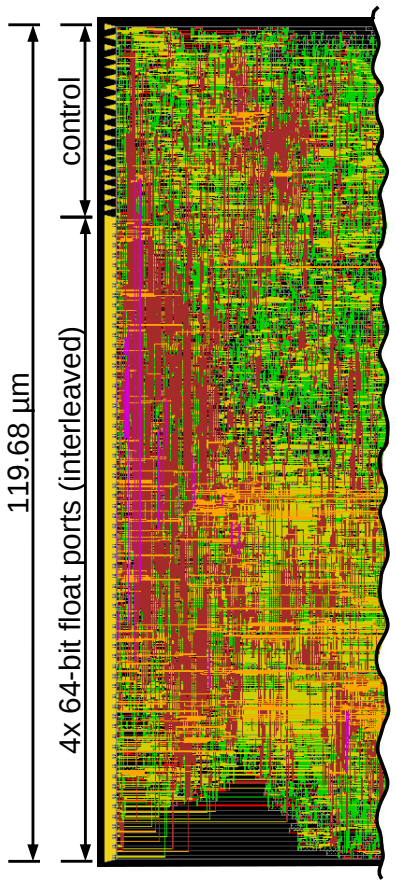
4. FMA Unit Back-End

For the Back-End implementation, we chose a recent technology: Global Foundries' 22 nm FDSOI process. It applies an additional insulator layer to remove the diodes between drain/source and the substrate. Furthermore, the channel is fully depleted, i.e. it is not weakly doped, to reduce leakage current. The insulator layer acts like a back-gate, which can be used to modify the transistor's threshold voltage. A back-gate bias voltage generator can later be used to apply a voltage to the back-gate, called Body Bias (BB). This allows to tune the circuit for either more performance or lower leakage or compensate process corners. The latter already emphasizes that this bias needs to be treated like process, voltage and temperature in static timing analysis. The GF22FDX process offers four types of transistors for different applications: high (HVT), regular (RVT), low (LVT), and super low (SLVT) threshold voltage devices. Since we are looking for performance, we decided to use the SLVT standard cell library.

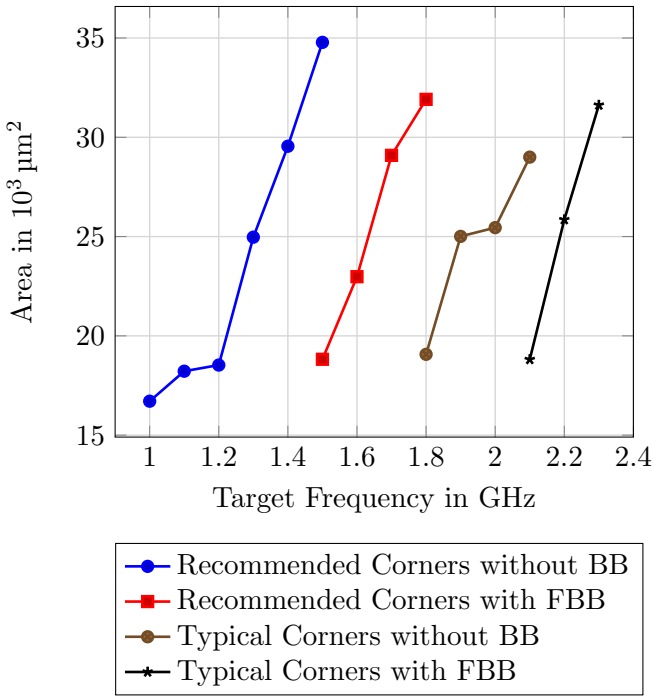
4.1. Synthesis Results

The floorplan was kept rather simple for this first implementation. We only defined the height to be 119.68 µm. The length was adjusted to yield a utilization of 80%. A more detailed placement will be part of future work, when more submodules are described at a lower abstraction level. This will allow for more control about what is synthesized. Pin placement was done with a later application in a RISC-V processor implementation in mind. RISC-V suggests a dedicated floating-point register file, which will need three read- and one write-port to provide the operands for fused operations. Assuming the register file will be located left of the FPU in a pipeline, we placed the operand and result pins on the left side in an interleaved manner using metal layers 3 to 6 and a spacing of 0.35 µm. The remaining pins are also placed on the left side with a spacing of 1.4 µm on metal 3 following the pins of `port_a`. This is shown in Fig. 3a. Depending on the exact register file size, this spacing may need to be changed in the future. To get a realistic timing we already performed scan insertion for this first synthesis run. This replaces all Flip-Flops (FFs) with Scan Flip-Flops (SFFs) that have a multiplexer in the datapath to switch between the regular input (D) and the scan input (SI). The additional multiplexer delay reduces the time available for other logic, but a scan chain is needed for chip testing. The effect of the clock distribution was also considered by performing Clock Tree Synthesis (CTS). This adds a buffer tree to the design to distribute the clock to all clock inputs and assures that the rising clock edge reaches every FF within a defined time window. Subsequent to CTS the design was routed. After routing the timing was met for a cycle time of 666 ps, i.e. 1.5 GHz over all recommended implementation corners without using FBB. Figure 3b shows more details of our results.

The synthesis results for the lower frequencies (blue curve) were obtained using the recommended corners for setup and hold analysis. These are slow (SS) and fast (FF) process, 10% voltage deviation around the nominal voltage of 0.8 V and a temperature of −40 °C and 125 °C. From these recommended corners the tools identified the combination (SS, 0.72 V, 125 °C, RC max) to be most timing critical. The currently only partially optimized design suffers from a significant area increase with rising target frequency. Figure 3b shows that the area roughly doubles from 1.2 GHz to 1.5 GHz. Despite 1.5 GHz being our target frequency for now, we conducted some tests for higher frequencies. For a real design, we could apply a FBB to increase the performance. So we switched the corners to the corresponding recommended corners with



a) Pin placement



b) Area over target frequency

Figure 3. Synthesis results

FBB. This allowed us to reach frequencies up to 1.8 GHz (red curve). To see where the design performs typically, we synthesized with typical (TT) corners only (brown curve) and also with their forward body biased versions (black curve). Typical corners are available for 25 °C and 85 °C. Here the tools identified (TT, 0.8 V, 85 °C, RC nominal) to be most timing critical. This allowed us to reach up to 2.3 GHz. The tools used were Cadence Genus and Innovus.

Another observation is, that we can reach higher frequencies with only typical corners, than with the recommended corners using FBB. This shows it is not possible, at least for this design, to compensate a worst case corner completely by using FBB.

4.2. Power Analysis

For all points presented in Fig. 3b, a power analysis based on a value change dump (vcd) file containing stimuli was conducted. The stimuli were generated with a modified test from our verification environment using Cadence Xcelium. The testbench contained the netlist, derived after all the previously described synthesis steps where executed, and a clock signal of the corresponding target frequency. As a side effect we also got some confidence in the synthesis procedure by running some stimuli through the implemented netlist. The test itself applied new operands every clock cycle and performed a random operation with a random rounding mode. The total power consumption for every design is shown in Fig. 4. The values were calculated for the (TT, 0.8 V, 85 °C, RC nominal) corner for all points, with the ones implemented with FBB

having their power determined with the corresponding FBB corner. We chose this corner for the power analysis, since all parameters are closest to real operating conditions. The clock tree makes up between 0.77 % and 1.56 % of the total power from high to low target frequencies, which seems plausible for a small design. The power analysis was done with Cadence Innovus/Voltus.

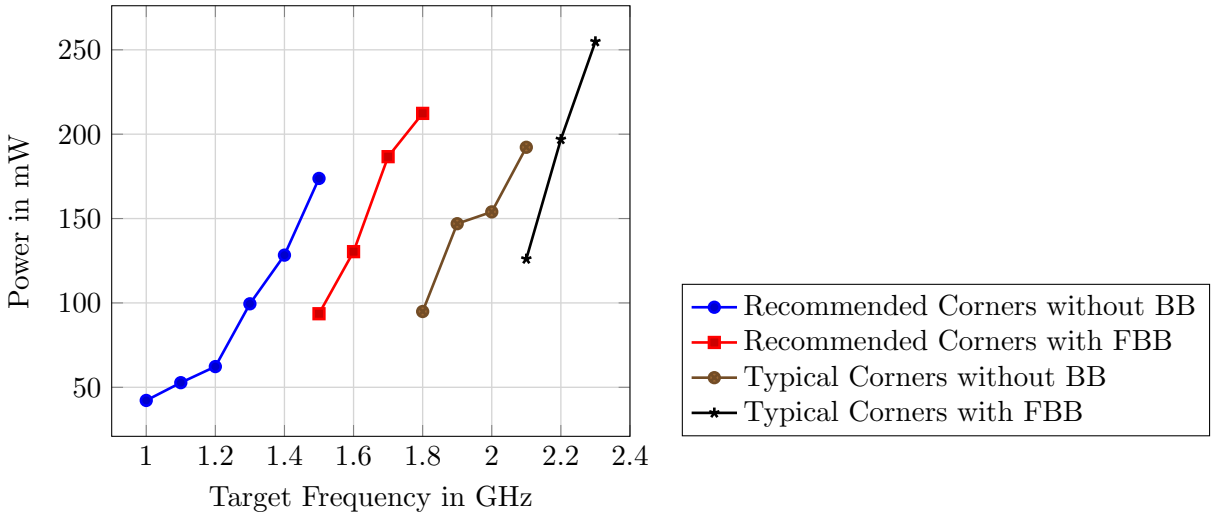


Figure 4. Power over target frequency for different corner setups

Figure 4 shows that the power scales with area and frequency. Figure 5 shows the power consumed by the 2.3 GHz implementation operating at frequencies from 1.0 GHz to 2.3 GHz. The linear rise of power with frequency is expected, but the values also show that a faster design, which uses more area, also consumes more power at lower operating frequencies than a design implemented for that particular target frequency. Besides the total power (black curve), Fig. 5 also shows the three parts which make up the total power. There is the switching power representing the loading/unloading of nets and the power used internally in the standard cells. They make up the linear part. The third part (brown curve) is the leakage power, which is constant at 33.6 mW over the operating frequency. This high leakage current is caused by using SLVT standard cells and FBB.

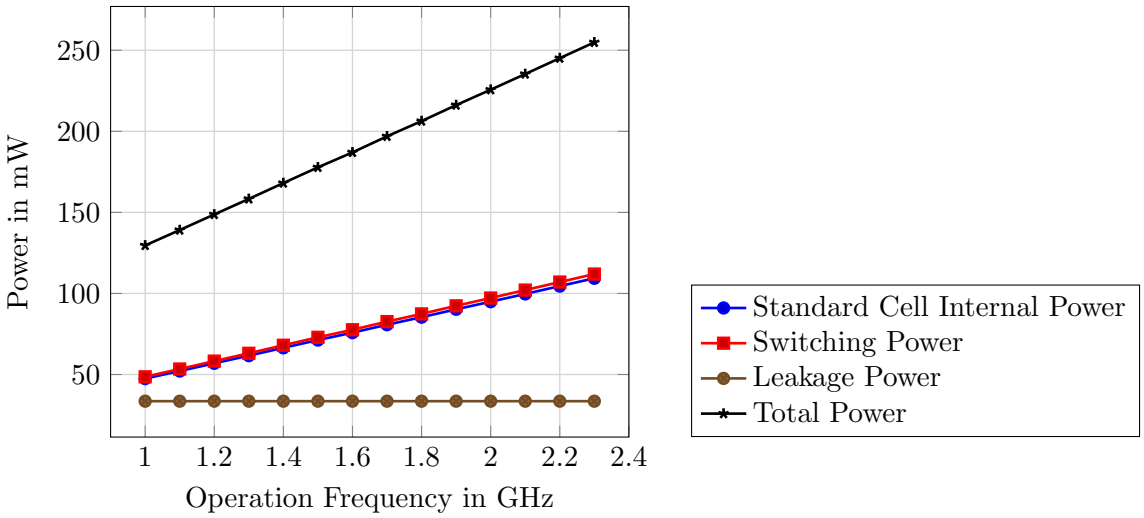


Figure 5. Power of the 2.3 GHz implementation over operation frequency

To evaluate our synthesis results we compared them to [13]. Table 2 compares the two closest FMA implementations in terms of the metrics used in [13]. We calculated the performance of our design by assuming two floating-point operations per clock cycle, i.e. assuming the maximum throughput possible. This was also done in [13]. Their design runs with 1.81 GHz in a 45 nm technology using low threshold devices and six pipeline stages. Compared to our synthesis result at 1.8 GHz for a typical process with super-low threshold devices, we have a similar power per performance but are roughly a factor of 3 smaller in terms of area per performance. The latter is contributed to technology scaling. The former is probably due to the low pipeline depth of three versus six. A lower number of pipeline stages makes it harder to achieve timing, thus requires the synthesis tool to use additional logic to fit the combinational logic into the cycle time. Furthermore, our design is not optimized for power in any way yet.

Table 2. Comparison of our synthesis results with [13]

Property	45 nm FMA [13]	Our 22 nm Design
V_{th}	low	super low
V_{DD} in V	0.9	0.8
Pipeline Depth	6	3
Frequency in GHz	1.81	1.8
Area in µm^2	49839	19066
W/GFLOPS	0.0253	0.0264
mm^2/GFLOPS	0.0145	0.0053
W/mm^2	1.75	4.98

Another interesting fact is seen in Fig. 6, which shows power per area over target frequency. Firstly, power density scales linearly with frequency as expected. But the second observation is that using FBB keeps the power density constant, whereas going from slow to typical corners reduces power more than one would expect from the area shrink alone. This shows again, that FBB is not enough to compensate worst case corners.

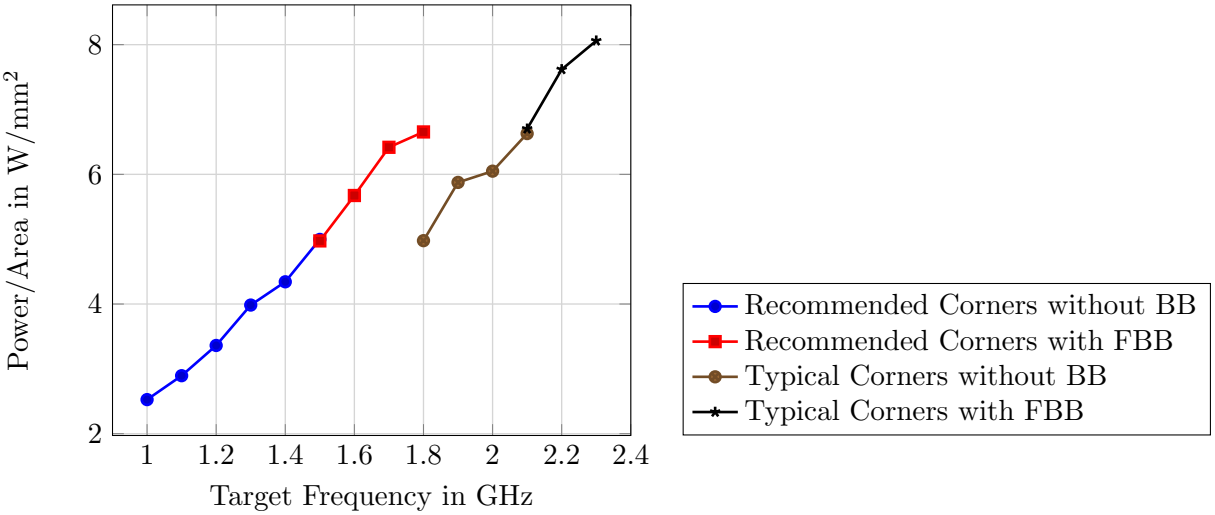


Figure 6. Power/Area over target frequency for different corner setups

Conclusion

This paper presents a design of a RISC-V- and IEEE 754-2008-conform FMA unit, which passed the complete ASIC design flow. This work's target is to lay the foundation for a high-speed FPU. Although it is still work in progress, it reaches 1.8 GHz using FBB - which is faster than the HardFloat of the U54 Rocket chip [7]. However, we do not have information on how fast other FPUs could be implemented standalone, without the corresponding CPU core. In terms of power, we still have room to improve, but power efficiency was not our goal. Still, we will certainly have a closer look at it in the future.

Additional to design and Back-End flow, we ensured functional correctness by performing a verification in which we checked the FMA unit against Intel's FPU and the Softfloat reference model. We also performed a small number of tests on the gate level during generation of the vcd file used for the power analysis.

References

1. Intel C++ Intrinsics Reference. `http://www.info.univ-angers.fr/pub/richer/ens/l3info/ao/intel_intrinsics.pdf` (2007), accessed: 2019-06-21

2. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 pp. 1–70 (2008), DOI: 10.1109/IEEESTD.2008.4610935

3. Aharoni, M., Asaf, S., Fournier, L., Koifman, A., Nagel, R.: FPgen - a test generation framework for datapath floating-point verification. In: Eighth IEEE International High-Level Design Validation and Test Workshop 2003, 12-14 November 2003, San Francisco, California, USA. pp. 17–22 (2003), DOI: 10.1109/HLDVT.2003.1252469

4. Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., et al.: The Rocket Chip Generator. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2016-17 (2016)

5. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: Constructing hardware in a Scala embedded language. In: The 49th Annual Design Automation Conference 2012, DAC 2012, 3-7 June 2012, San Francisco, California, USA. pp. 1212–1221 (2012), DOI: 10.1145/2228360.2228584

6. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together – Formal verification of the VAMP. International Journal on Software Tools for Technology Transfer 8(4), 411–430 (2006), DOI: 10.1007/s10009-006-0204-6

7. Celio, C., Chiu, P.F., Nikolic, B., Patterson, D., Asanović, K.: BOOM v2 an open-source out-of-order RISC-V core. `https://content.riscv.org/wp-content/uploads/2017/12/Wed0936-BOOM-v2-An-Open-Source-Out-of-Order-RISC-V-Core-Celio.pdf` (2017), accessed: 2019-06-21

8. Celio, C., Patterson, D.A., Asanović, K.: The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2015-167 (2015)

9. Clarke, E.M., German, S.M., Zhao, X.: Verifying the SRT division algorithm using theorem proving techniques. In: Computer Aided Verification. pp. 111–122. Springer Berlin Heidelberg, Berlin, Heidelberg (1996), DOI: 10.1007/3-540-61474-5_62

10. Clarke, E.M.: The Birth of Model Checking. In: 25 Years of Model Checking: History, Achievements, Perspectives, pp. 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), DOI: 10.1007/978-3-540-69850-0_1

11. Conti, F., Rossi, D., Pullini, A., Loi, I., Benini, L.: Energy-efficient vision on the PULP platform for ultra-low power parallel computing. In: 2014 IEEE Workshop on Signal Processing Systems, SiPS 2014, 20-22 October 2014, Belfast, United Kingdom. pp. 1–6 (2014), DOI: 10.1109/SiPS.2014.6986099

12. Dongarra, J.J.: The LINPACK Benchmark: An explanation. In: Supercomputing, ICS 1987, 1st International Conference Athens, Greece, June 8–12, 1987. pp. 456–474. Springer Berlin Heidelberg, Berlin, Heidelberg (1988), DOI: 10.1007/3-540-18991-2_27

13. Galal, S., Horowitz, M.: Energy-Efficient Floating-Point Unit Design. IEEE Transactions on Computers 60(7), 913–922 (2011), DOI: 10.1109/TC.2010.121

14. Harrison, J.: Formal verification of ia-64 division algorithms. In: Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000 Portland, OR, USA, August 14–18, 2000. pp. 233–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2000), DOI: 10.1007/3-540-44659-1_15

15. Hauser, J.: The softfloat and testfloat packages. `http://www.jhauser.us/arithmetic/` (2015), accessed: 2019-06-21

16. Li, L.: PULP-Platform - FPU. `https://github.com/pulp-platform/fpu` (2017), accessed: 2019-06-21

17. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: Top 500 list. `https://www.top500.org/lists/2018/11/` (2018), accessed: 2019-06-21

18. Muller, J.M., Brisebarre, N., De Dinechin, F., Jeannerod, C.P., Lefevre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of floating-point arithmetic. Springer Science & Business Media (2009)

19. Waterman, A., Lee, Y., Hauser, J.: Berkeley Hardware Floating-Point Units. `https://github.com/ucb-bar/berkeley-hardfloat` (2018), accessed: 2019-06-21

20. Waterman, A.S.: Design of the RISC-V Instruction Set Architecture. Ph.D. thesis, EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2016-1 (2016)

DOI: 10.14529/jsfi190206

Fully Implicit Time Stepping Can Be Efficient on Parallel Computers

Brandon Cloutier[1], *Benson K. Muite*[2], *Matteo Parsani*[3]

Benchmarks in high performance computing often involve a single component used in the full solution of a computational problem, such as the solution of a linear system of equations. In many cases, the choice of algorithm, which can determine the components used, is also important when solving a full problem. Numerical evidence suggests that for the Taylor–Green vortex problem at a Reynolds number of 1600, a second order implicit midpoint rule method can require less computational time than the often used linearly implicit Carpenter–Kennedy method for solving the equations of incompressible fluid dynamics for moderate levels of accuracy at the beginning of the flow evolution. The primary reason is that even though the implicit midpoint rule is fully implicit, it can use a small number of iterations per time step, and thus require less computational work per time step than the Carpenter–Kennedy method. For the same number of timesteps, the Carpenter–Kennedy method is more accurate since it uses a higher order timestepping method.

Keywords: incompressible Navier–Stokes equations, parallel computing, spectral methods, time stepping.

Introduction

Benchmarks in parallel computing are often micro-benchmarks or computationally expensive components of full applications, such as solvers for linear systems. It is often the case that the choice of numerical method can be as important as the optimization of the component subroutines. To be able to compare different choices of numerical methods, full application benchmarks are also very useful. The International Workshops on High Order Computational Fluid Dynamics methods [7, 12] are a conference series with the aim of comparing the speed and accuracy of computational fluid dynamics software for solving particular well defined problems. One of the considered cases is the Taylor–Green vortex at a Reynolds number of 1600. This is a case where there is a vortex instability which is difficult to compute correctly using low resolution or a poor numerical method [24]. While there are a wide range of spatial discretization methods suitable for solving the Navier–Stokes equations, Fourier spectral methods are a class of methods where the choice of time stepping procedure on the effectiveness of solving a partial differential equation can be easily investigated [20]. By comparing two different time stepping regimes, one can also determine whether discretization methods which have conserved discrete analogues of the continuum conserved quantities are well suited for approximating turbulent flows, and will be useful in engineering applications where time and cost to solution are important.

Following this introduction, Section 1 introduces the incompressible Navier–Stokes equations; the numerical algorithms which use the Fast Fourier transform as a component are described in the Section 2. A description of the massively parallel computational platform used is in Section 3. The results of the computational experiments are described in Section 4. This is then followed by the conclusion.

[1]University of Michigan, Ann Arbor, Michigan, USA
[2]Tartu Ülikool, Tartu, Estonia
[3]King Abdullah University of Science and Technology (KAUST), Computer Electrical and Mathematical Science and Engineering Division (CEMSE), Extreme Computing Research Center (ECRC), Thuwal, Saudi Arabia

1. Incompressible Navier–Stokes Equations

The incompressible Navier–Stokes equations written in dimensionless form are:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla \cdot p + \frac{1}{\mathrm{Re}} \Delta \mathbf{u}, \qquad \nabla \cdot \mathbf{u} = 0, \tag{1}$$

where $\mathbf{u} \in \mathbb{R}^3$ is the velocity vector and $p \in \mathbb{R}$ is the pressure. Here Re denotes the Reynolds number that characterizes the flow and it is defined as

$$\mathrm{Re} = \frac{\rho_{ref} |\mathbf{u_{ref}}| \mathbf{L_{ref}}}{\mu_{ref}}, \tag{2}$$

where ρ_{ref}, $|\mathbf{u_{ref}}|$, L_{ref}, μ_{ref} and are the (constant) density of the fluid, the modulus of a representative velocity, a characteristic length and the dynamic viscosity, respectively.

2. The Numerical Method

A Fourier spectral method is used to perform a direct numerical simulation of the Taylor–Green vortex for the incompressible Navier–Stokes equations. For the Taylor–Green vortex flow, given the periodic square box $[-m\pi, m\pi]^3$ (which defines the computational domain) and the initial velocity field, the representative length scale and velocity module are set equal to the multiple m of the "standard" box width 2π (i.e., $L_{ref} = m$) and the maximum velocity component of the initial flow field.

The parallel codes are available at [9, 10]. They use MPI and are similar to the example programs available at [4]. The library 2DECOMP&FFT is used for the domain decomposition and the parallel fast Fourier transforms [21].

In both implicit midpoint rule (IMR) and Carpenter–Kennedy [5] (CK) time stepping schemes, the time advancement is done in Fourier space and the nonlinear terms are calculated by transforming to real space, multiplying and then transforming back to Fourier space. Following [6], we do not de-alias our schemes because the simulations are fully resolved. Visualization is done using Octave [15], Matlab, Python (Matplotlib [19]), Paraview [2] and VisIt [8].

2.1. Implicit Midpoint Rule

The implicit midpoint rule is

$$\begin{aligned} &\frac{\mathbf{u}^{n+1,j+1} - \mathbf{u}^n}{\delta t} + \frac{\mathbf{u}^{n+1,j} + \mathbf{u}^n}{2} \cdot \nabla \left(\frac{\mathbf{u}^{n+1,j} + \mathbf{u}^n}{2} \right) \\ &= \frac{\nabla \left[\Delta^{-1} \left(\nabla \cdot \left[(\mathbf{u}^{n+1,j} + \mathbf{u}^n) \cdot \nabla (\mathbf{u}^{n+1,j} + \mathbf{u}^n) \right] \right) \right]}{4} + \Delta \frac{\mathbf{u}^{n+1,j+1} + \mathbf{u}^n}{2Re}, \end{aligned} \tag{3}$$

where j is the iterate and n is the timestep. Note that the implementation of the IMR in this study uses fixed point iteration with the stopping critera that the change between successive iterations is less than 10^{-10} in the sum of the l^∞ norms of the components of the individual flow field components. The method requires three levels of storage for $\mathbf{u}^{n+1,j+1}$, $\mathbf{u}^{n+1,j}$ and $\mathbf{u}^n$, and uses 15 Fast Fourier Transforms per iteration.

The IMR method also has discrete analogues for energy and enstrophy dissipation:

$$\frac{1}{2} \int_\Omega \frac{\mathrm{d}(\mathbf{u} \cdot \mathbf{u})}{\mathrm{d}t} = -\frac{1}{Re} \int_\Omega \nabla \mathbf{u} \cdot \nabla \mathbf{u}, \tag{4}$$

$$\int_\Omega \frac{\mathbf{u}^{n+1}\cdot\mathbf{u}^{n+1}-\mathbf{u}^{n}\cdot\mathbf{u}^{n}}{2\delta t} = -\frac{1}{4Re}\int_\Omega \nabla\left(\mathbf{u}^{n+1}+\mathbf{u}^{n}\right)\cdot\nabla\left(\mathbf{u}^{n+1}+\mathbf{u}^{n}\right), \tag{5}$$

$$\frac{1}{2}\int_\Omega \frac{\mathrm{d}(\omega\cdot\omega)}{\mathrm{d}t} = -\frac{1}{Re}\int_\Omega \nabla\omega\cdot\nabla\omega, \tag{6}$$

$$\int_\Omega \frac{\omega^{n+1}\cdot\omega^{n+1}-\omega^{n}\cdot\omega^{n}}{2\delta t} = -\frac{1}{4Re}\int_\Omega \nabla\left(\omega^{n+1}+\omega^{n}\right)\cdot\nabla\left(\omega^{n+1}+\omega^{n}\right). \tag{7}$$

2.2. Carpenter–Kennedy Method

The CK method consists of splitting the equation into linear and nonlinear parts,

$$\mathbf{l}(\mathbf{u}) = \frac{1}{\mathrm{Re}}\Delta\mathbf{u}, \tag{8}$$

$$\mathbf{g}(\mathbf{u}) = -\mathbf{u}\cdot\nabla\mathbf{u} + \nabla\left[\Delta^{-1}\left(\nabla\cdot[\mathbf{u}\cdot\nabla\mathbf{u}]\right)\right] \tag{9}$$

and then solving the linear part implicitly, and the nonlinear part explicitly, using the steps detailed in Algorithm 1, where in line 5

$$\alpha = [0.0, 0.1496590219993, 0.3704009573644, 0.6222557631345, 0.9582821306748, 1.0],$$

$$\beta = [0.0, -0.4178904745, -1.192151694643, -1.697784692471, -1.514183444257]$$

and in line 6

$$\gamma = [0.1496590219993, 0.3792103129999, 0.8229550293869, 0.6994504559488, 0.1530572479681].$$

The algorithm requires two levels of storage h and u and uses 70 FFTs per timestep. Thus, if the IMR method requires 4 or fewer iterations per timestep, it can require less time to complete than the CK method.

Algorithm 1 The CK time stepping algorithm for the 3D incompressible Navier-Stokes equations

1: **input** $\mathbf{u^n}$
2: $\mathbf{u} = \mathbf{u^n}$
3: $\mathbf{h} = \mathbf{0}$
4: **for** $k = 1$ **to** 5 **do**
5: $\quad \mathbf{h} \leftarrow \mathbf{g}(\mathbf{u}) + \beta_k \mathbf{h}$ $\mu \leftarrow 0.5\delta t\,(\alpha_{k+1} - \alpha_k)$
6: $\quad$ **Solve** $\mathbf{v} - \mu\mathbf{l}(\mathbf{v}) = \gamma_k \delta t \mathbf{h} + \mu\mathbf{l}(\mathbf{u})$ **for** $\mathbf{v}$
7: $\quad \mathbf{u} \leftarrow \mathbf{v}$
8: **end for**
9: $\mathbf{u}^{n+1} = \mathbf{u}$
10: **return** $\mathbf{u}^{n+1}$

3. Computational Platform

Numerical experiments have been performed on a variety of computational platforms, which are listed in the acknowledgements. The results reported here were obtained on Hazelhen [18] a Cray XC 40 supercomputer with dual Intel® Xeon® CPU E5-2680 v3 (30M Cache, 2.50 GHz) processors per node. This supercomputer uses an Aries interconnect [17]. As this computer has a network where congestion effects can change runtime, reported results were run on 768 cores for which repeated short runs showed that the typical standard deviation in the runtime was 20% [3], see also Fig. 10. Access to a computer system with a network that isolates communication for a particular job to a particular subnetwork (such as found on supercomputers with torus networks) was not available for this study.

The source codes [9, 10] were compiled with GCC GNU Fortran compilers using the Cray provided wrappers (ftn) and optmization flag -O3. 2DECOMP&FFT [21] was used as the parallel Fourier transform and domain decomposition library, with FFTW 3.3.4.7 as the one dimensional Fast Fourier transform library.

4. Results

4.1. Verification

To ensure verifiability of the computed results, the benchmark case requires production of data for the kinetic energy (Fig. 1), kinetic energy dissipation rate (Fig. 2) and enstrophy (Fig. 3) evolution for the Taylor–Green Vortex between times of 0 and 10. Also required is a plot of the midplane vorticity (Fig. 4) at a time of 8, when the enstrophy and kinetic energy dissipation rates are at their peaks. Reference values for these are given by the organizers to allow for comparison with submitted solutions. Many previous submissions have typically reported solutions with 2-3 digits of accuracy. The programs and data to produce all figures except Fig. 4 are in [11].

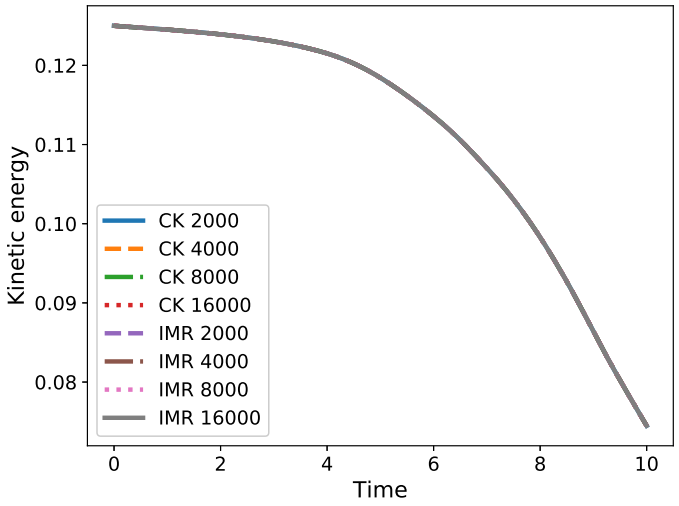


Figure 1. Kinetic Energy. Legend indicates time stepping scheme and number of timesteps

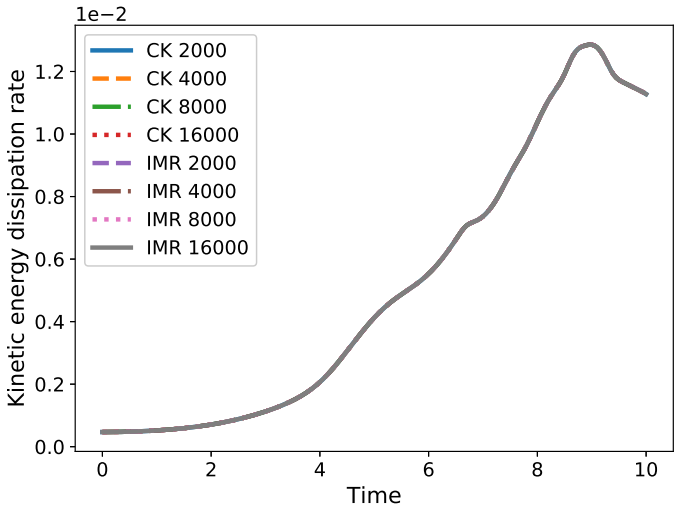


Figure 2. Kinetic energy dissipation rate. Legend indicates time stepping scheme and number of timesteps

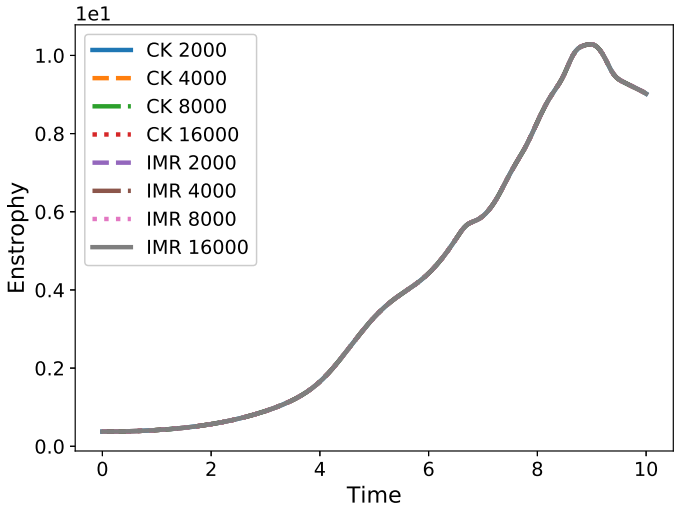


Figure 3. Enstrophy. Legend indicates time stepping scheme and number of timesteps

4.2. Comparison between the Implicit Midpoint Rule and Carpenter–Kennedy Methods

The IMR requires 15 fast Fourier transforms per iteration, while the CK method requires 70 iterations per timestep. Thus for a single timestep, if 4 or fewer iterations are needed, the IMR method will require less time to solution than the CK method. The IMR method is a second order method, while the CK method is a third order method, though for high Reynolds number flows, behaves like a fourth order method. Nevertheless, the IMR method should be good at capturing statistical behavior of turbulent flows.

Figure 5 shows the number of fixed point iterations required for the IMR during the computation, and Tab. 1 shows the average number of fixed point iterations, as well as the total core hours required for a computation. The number of iterations is smaller when the timestep is small and increases as the enstrophy (Fig. 3) and kinetic energy dissipation rates (Fig. 2) in the flow increase. Figure 6 and 9 show that the IMR method gives levels of accuracy of 10^{-3} for enstrophy and 10^{-6} for the kinetic energy dissipation rate, and the CK method gives levels of accuracy of 10^{-6} for enstrophy and 10^{-9} for the kinetic energy dissipation rate.

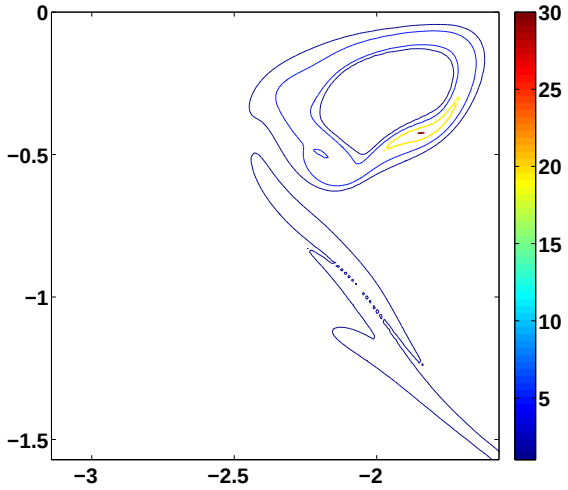


Figure 4. Isocontours of the midplane vorticity at a time of 8, due to symmetry only a quarter of the midplane is shown

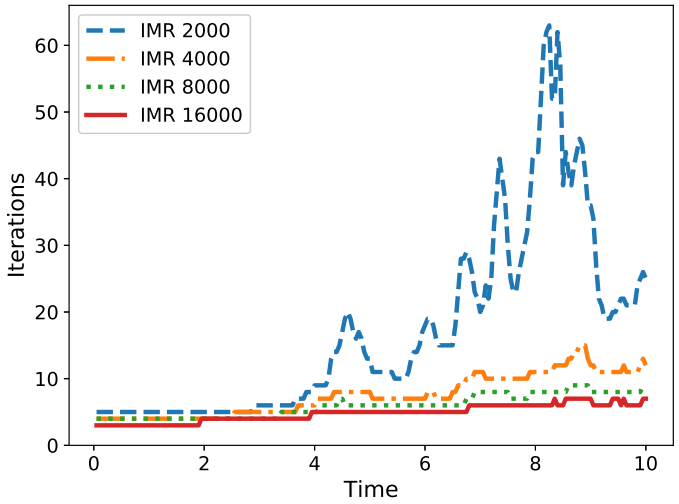


Figure 5. Iterations required for convergence to the required tolerance for the IMR scheme during the temporal flow field evolution. Legend indicates the number of timesteps

Table 1. Performance on Hazelhen [18]. All computations are done from a time of 0 to a time of 10 on 768 cores for a 512^3 discretization

Method	Time steps	Time(s)	Core hours	FFTs/Timestep
IMR	2000	10369	2212.1	254.8
CK	2000	2513	536.1	70
IMR	4000	8295	1769.6	112.7
CK	4000	5176	1104.2	70
IMR	8000	13199	2815.8	88.9
CK	8000	10722	2287.4	70
IMR	16000	21918	4675.8	72.45
CK	16000	20608	4396.4	70

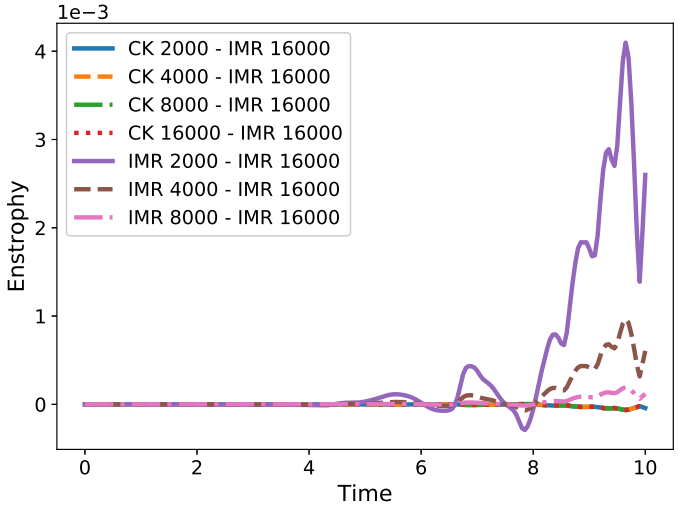


Figure 6. Differences between the enstrophy computed with the IMR scheme using 16000 timesteps and CK and IMR schemes. Legend indicates the number of timesteps

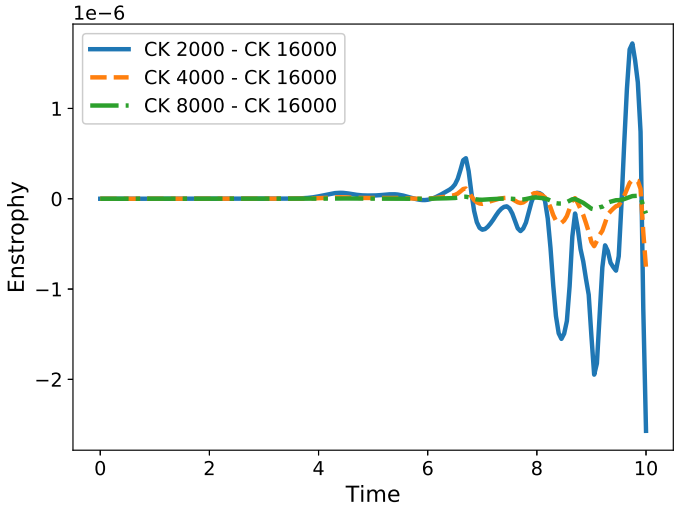


Figure 7. Differences between the enstrophy computed with CK schemes. Legend indicates the number of timesteps

Finally, Fig. 10 shows that for the first 20 timesteps of a 2000 timestep run, the IMR and CK methods have the same runtime.

Conclusions

In the initial phase of the Taylor–Green vortex flow, the second order implicit midpoint rule is efficient for moderate accuracy simulations because it requires only a few fixed point iterations to converge. Despite the fact that the implicit midpoint method preserves the energy dissipation structure of the equations, the current results show that for the time scale and Reynolds numbers considered here, the higher order Carpenter–Kennedy method is more accurate at capturing the global kinetic energy and enstrophy evolutions. Structure preserving schemes, like the implicit midpoint rule are often used in computer graphics simulations [16]. These can be coupled with spatial discretization methods that have lower communication requirements than the fast Fourier

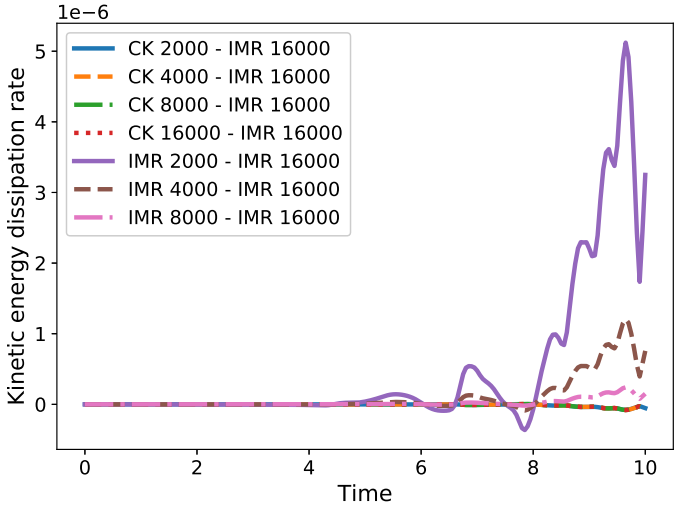


Figure 8. Differences between the kinetic energy dissipation rates computed with the IMR and CK schemes. Legend indicates the number of timesteps

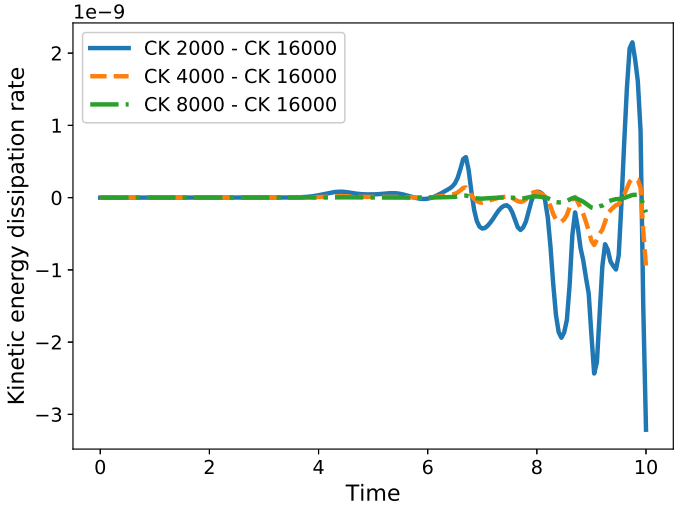


Figure 9. Differences between the kinetic energy dissipation rates computed with the CK method. Legend indicates the number of timesteps

transform, such as finite difference or finite element methods, to give moderate accuracy results with relatively low computation times.

Relying on classical schemes may be inappropriate, especially for computationally costly simulations where statistical reproducibility but not point wise accuracy is required. Thus in addition to semi-implicit schemes such as the Carpenter–Kennedy method, fully implicit schemes should also be considered as they may be computationally efficient [23]. This is because for time evolutionary schemes (as opposed to stationary problems as considered in other studies [1, 13, 22]), a good initial iterate obtained from the numerical approximation at the previous time step can make the number of iterations required for convergence of the iterative scheme small. In the atmospheric simulations [25] it has also been observed that implicit schemes may give a faster time to solution than explicit schemes, despite having lower scalability and a lower floating point efficiency.

For many computer scientists, algorithm/subroutine optimization is a common task, but full application optimization typically also requires domain specific knowledge. At present, it

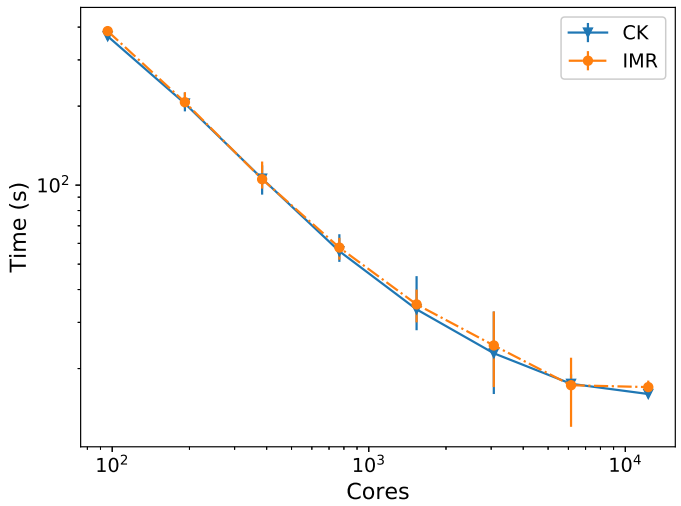


Figure 10. Strong Scaling for the first 20 timesteps of a 2000 timestep run modified from [3]. Error bars represent maximum and minimum runtimes

is challenging to find studies that combine both domain specific and high performance computing knowledge. Domain specific community benchmarking efforts, such as the International Workshop on High Order Computational Fluid Dynamics serve as a very useful complement to traditional high performance computing benchmarks such as Linpack [14]. Such efforts will become much more relevant in the future as most scientific computing will necessarily be parallel.

Acknowledgements

We thank the reviewers for their constructive comments, and Koen Hillewaert and David Ketcheson for helpful advice. We thank the participants of the 2013 HiOCFD workshop [12] for feedback on an earlier version of this work that enabled greater examination of the numerical results. BKM thanks Arieh Iserles for pointing out the geometric properties of the implicit midpoint rule and Charles Doering, José Gracia, Hans Johnston, Ning Li, Peter Van Keken, Divakar Viswanath and Jared Whitehead for helpful discussion. BKM was partially supported by HPC Europa 3 (INFRAIA-2016-1-730897). The computational resources used to build and test the programs were:

- Flux operated at the University of Michigan.
- Keeneland initial delivery system operated by Georgia Tech and the National Institute for Computational Sciences.
- Noor operated by KAUST information technology services.
- Shaheen and Shaheen II operated by the KAUST Supercomputing Laboratory.
- Hazelhen and Kabuki at HLRS.

Flow field visualization was aided by an extended collaborative support allocation from the extreme digital science and discovery environment (XSEDE) which is supported by National Science Foundation grant number OCI-1053575. We thank Mark Vanmoer for help with this.

References

1. Adams, M.F., Brown, J., Shalf, J., Straalen, B.V., Strohmaier, E., Williams, S.: HPGMG 1.0: A benchmark for ranking high performance computing systems. Tech. Rep. LBNL 6630E, Lawrence Berkeley National Laboratory, Berkeley, California, USA (2014). https://escholarship.org/uc/item/00r9w79m, accessed: 2019-07-17

2. Ahrens, J., Geveci B., Law C.: ParaView: An End–User Tool for Large Data Visualization, In the Visualization Handbook. Edited by Hansen C.D. and Johnson C.R., Elsevier (2005). http://www.paraview.org/, accessed: 2019-06-24

3. Aseeri, S., Muite, B.K., Takahashi, D.: Reproducibility in Benchmarking Parallel Fast Fourier Transform based Applications. In: Companion of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE '19, 7-11 April 2019, Mumbai, India. pp. 5–8. ACM (2019), DOI: 10.1145/3302541.3313105

4. Balakrishnan, S., Bargash, A.H., Chen, G., Cloutier, B., Li, N., Malicke, D., Muite, B.K., Quell, M., Rigge, P., San Roman Alerigi, D., Solimani, M., Souza, A., Thiban, A.S., West, J., van Moer, M.: Parallel Spectral Numerical Methods. http://en.wikibooks.org/wiki/Parallel_Spectral_Numerical_Methods, accessed: 2019-06-24

5. Carpenter, M.H., Kennedy, C.A.: Fourth–Order 2N–Storage Runge–Kutta schemes, NASA Langley Research Center Technical Memorandum 109112. https://ntrs.nasa.gov/search.jsp?R=19940028444 (1994), accessed: 2019-07-17

6. Canuto, C., Hussaini, M.Y., Quarteroni, A., Zang, T.A.: Spectral methods fundamentals in single domains, Springer (2010), DOI: 10.1007/978-3-540-30726-6

7. Cenaero: Fifth International Workshop on High–Order CFD Methods. https://how5.cenaero.be/, accessed: 2019-06-24

8. Childs, H., Brugger, E., Bonnell, K., Meredith, J., Miller, M., Whitlock, B., Max, N.: A contract–based system for large data visualization, IEEE Visualization 2005, 190–198, (2005). https://wci.llnl.gov/codes/visit/, accessed: 2019-06-24

9. Cloutier, B.: MPI Fortran Implicit Midpoint Rule Incompressible Navier–Stokes Solver. https://github.com/bcloutier/PSNM/blob/master/NavierStokes/Programs/NavierStokes3dFortranMPI/navierstokes_IMR.f90, accessed: 2019-06-24

10. Cloutier, B.: MPI Fortran Carpenter-Kennedy Incompressible Navier–Stokes Solver. https://github.com/bcloutier/PSNM/blob/master/NavierStokes/Programs/NavierStokes3dFortranMPI/navierstokes.f90, accessed: 2019-06-24

11. Cloutier, B., Muite, B.K., Parsani, M.: Fully Implicit Time Stepping can be Efficient on Parallel Computers [Data set], Zenodo, DOI: 10.5281/zenodo.2667709

12. Deutsches Zentrum für Luft- und Raumfahrt (DLR): Second International Workshop on High–Order CFD Methods. https://www.dlr.de/as/desktopdefault.aspx/tabid-8170/13999_read-35550/, accessed: 2019-06-24

13. Dongarra, J., Heroux, M., Luszczek, P.: HPCG benchmark: a new metric for ranking high performance computing systems. The International Journal of High Performance Computing Applications 30(1), 3–10 (2015), DOI: 10.1177/1094342015593158

14. Dongarra, J., Luszczek, P., Petitet, A.: The LINPACK benchmark: Past, present and future. Concurrency and Computation: Practice and Experience 38(9), 803–820 (2003), DOI: 10.1002/cpe.728

15. Eaton, J. W., Bateman, D., Hauberg, S., Wehbring, R.: GNU Octave version 4.2.1 manual: a high–level interactive language for numerical computations. `https://www.gnu.org/software/octave/doc/v4.2.1/` (2017), accessed: 2019-06-24

16. Elcott, S., Tong, Y., Kanso, E., Schröder, P., Desbrun, M.: Stable, circulation–preserving, simplicial fluids. ACM Transactions on Graphics 26(1), 4 (2007), DOI: 10.1145/1189762.1189766

17. Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., Reinhard, J.: Cray Cascade: A scalable HPC system based on a Dragonfly network. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 1–9 (2012), DOI: 10.1109/SC.2012.39

18. Höchstleistungsrechenzentrum Stuttgart (HLRS): Hazelhen. `https://www.hlrs.de/systems/cray-xc40-hazel-hen/`, accessed: 2019-06-24

19. Hunter, J.D.: Matplotlib: A 2D graphics environment. Computing in Science and Engineering 9(3), 90–95 (2007), DOI: 10.1109/MCSE.2007.55

20. Ketcheson, D.I., Mortensen, M., Parsani, M., Schilling N.: More efficient time integration for Fourier pseudo–spectral DNS of incompressible turbulence, arXiv: 1810.10197v1, accessed: 2019-07-17

21. Li, N., Laizet, S.: 2DECOMP&FFT – A highly scalable 2D decomposition library and FFT interface, Cray User Group 2010, Edinburgh, UK. `http://www.2decomp.org/pdf/17B-CUG2010-paper-Ning_Li.pdf`, accessed: 2019-07-17

22. Müller, E.H., Scheichl, R., Vainikko, E.: Petascale solvers for anisotropic PDEs in atmospheric modelling on GPU clusters. Parallel Computing 50, 53–69 (2015), DOI: 10.1016/j.parco.2015.10.007

23. Parsani, M., Van den Abeele, K., Lacor, C. and Turkel, E.: Implicit LU–SGS algorithm for high–order methods on unstructured grid with p–multigrid strategy for solving the steady Navier–Stokes equations. Journal of Computational Physics 229(3), 828–850 (2009), DOI: 10.1016/j.jcp.2009.10.014

24. van Rees, W.M., Leonard, A., Pullin, D.I., Koumoutsakos, P.: A comparison of vortex and pseudo–spectral methods for the simulation of periodic vortical flows at high Reynolds numbers. J. of Computational Physics 230, 2794–2805 (2011), DOI: 10.1016/j.jcp.2010.11.031

25. Yang, C., Xue, W., Fu, H., You, H., Wang, X., Ao, Y., Liu, F., Gan, L., Xu, P., Wang, L., Yang, G., Zheng, W.: 10M–core scalable fully–implicit solver for nonhydrostatic atmospheric dynamics. In: SC'16: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 57–68 (2016), DOI: 10.1109/SC.2016.5

DOI: 10.14529/jsfi190207

Performance Limits Study of Stencil Codes on Modern GPGPUs

Ilya S. Pershin[1], *Vadim D. Levchenko*[2], *Anastasia Y. Perepelkina*[2]

We study the performance limits of different algorithmic approaches to the implementation of a sample problem of wave equation solution with a cross stencil scheme. With this, we aim to find the highest limit of the achievable performance efficiency for stencil computing.

To estimate the limits, we use a quantitative Roofline model to make a thorough analysis of the performance bottlenecks and develop the model further to account for the latency of different levels of GPU memory. These estimates provide an incentive to use spatial and temporal blocking algorithms. Thus, we study stepwise, domain decomposition, and domain decomposition with halo algorithms in that order. The knowledge of the limit incites the motivation to optimize the implementation. This led to the analysis of the block synchronization methods in CUDA, which is also provided in the text. After all optimizations, we have achieved 90% of the peak performance, which amounts to more than 1 trillion cell updates per second on one consumer level GPU device.

Keywords: stencil computations, parallel algorithm, GPU, CUDA, Roofline model.

Introduction

The theoretical performance peak of the modern GPU exceeds 10 TFLOPS for single precision, and this could be a trillion cell updates per second for a variety of stencil schemes. However, it is well known that this performance can not be achieved, since the stencil codes are not compute-bound [7]. The global memory throughput limit for modern GPU corresponds to approximately 1% of their peak computing performance. Furthermore, in the implementations of the stencil codes, other factors, such as data access overhead and latency, limit the performance.

In this paper we study the performance limits of different algorithmic approaches, applied to a sample problem, and aim to find the highest limit of the achievable performance efficiency for the stencil computing. For this, we develop the implementation using the accumulated experience of CUDA programming so as to minimize the performance losses. We use an advanced quantitative performance model to make a thorough analysis of the performance bottlenecks, and develop it further to account for the latency of different levels of GPU memory.

Increasing the performance efficiency of the stencil implementation is an intricate task, and multiple factors should be take into account. The cell update requires its data and the data from its neighbours to be accessed. Thus, each cell data is accessed multiple times during one time iteration, and the exact way this access is performed depends on the algorithm. The cache hierarchy is developed so as to accelerate the data accesses with high locality in time and space. To take advantage of it in the stencil computing, the tiling and blocking techniques are used [5, 8–10, 13, 14, 20, 27, 30]. These involve a modification of the data space traversal and decomposition of this task into subtasks that may be given to different processing units. Various polyhedral optimization techniques are based on this idea [6, 22].

Spatial tiling [5, 8–10, 13, 14, 20, 27, 30] involves only spatial directions and may be hierarchical to incorporate different levels of cache.

Temporal tiling [5, 8, 13, 14, 20, 27, 30] is the method to perform several cell updates on the same data portion that is located in the fast memory. After this, more data will be sent,

[1]Moscow Institute of Physics and Technology, Dolgoprudny, Russian Federation
[2]Keldysh Institute of Applied Mathematics RAS, Moscow, Russian Federation

although, there are less synchronization events. Hierarchical temporal tiling is a part of the Locally Recursive non-Locally Asynchronous (LRnLA) algorithms approach [25, 26].

The idea of the tiling applies to any system with hierarchical memory and levels of parallelism. It may be made cache-aware or cache-oblivious [12].

Concerning the stencil performance optimization on GPU, in the classical CUDA implementation [32] the tiling is inherent and is obtained by tuning the CUDA kernel parameters so as the data of the tile fits shared memory. This foreshadowed the trend for 2.5D blocking 1D streaming algorithm [13, 24, 31], which may be used in conjunction with temporal blocking [20]. The best performance of the applied stencil codes reaches about 30% of the peak theoretical performance [17, 18, 29].

The algorithmic optimization is often not enough for the best performance, the programming tools should be used with care. This includes the knowledge of the compiler optimization trends and hardware specifics. This becomes the center point of some optimization strategies [11, 24] and promotes the development of stencil code generation frameworks [4, 23, 28].

Since the stencil computing is considered as a memory-bound problem, the performance limits and bottlenecks in most of the aforementioned works are studied from the consideration of the ratio of the cell updates that may be performed per data access operation and the memory throughput. This analysis has become more convenient with the introduction of the Roofline model [21]. We use it in this work, and propose the modification to form an image of the latency Roofline. In sum, this work combines the use of the most advanced algorithms, CUDA implementation techniques and performance analysis to provide a model for maximizing the performance of the stencil code implementation on GPU. For this purpose, we choose the finite-difference cross-stencil scheme for 1D wave equation.

The article is organized as follows. Section 1 serves to introduce the kind of numerical computation that is considered for the current work. In sections 2, 3, and 4 we discuss the three algorithmic approaches, namely, stepwise algorithm, recursive domain decomposition, and recursive domain decomposition with halo, correspondingly. In each section, we present the description of the corresponding algorithm, details of its implementation, performance test results and the theoretical performance model in that order. At the end of sections 2 and 3, we evaluate the weak spots of each algorithm and propose the direction of further study. In the Conclusion, we summarize the achievements and propose the practical applications of the presented research.

1. Problem Statement and Cross Stencil

The following $(2r+2)$-point stencil computation is constructed (Fig. 1):

$$u_k^{n+1} = -u_k^{n-1} + \underbrace{\alpha_0 u_k^n + \alpha_{-1} u_{k-1}^n + \alpha_1 u_{k+1}^n + \ldots + \alpha_{-r} u_{k-r}^n + \alpha_r u_{k+r}^n}_{(2r+1)\quad \text{terms}}. \tag{1}$$

Here r is the stencil radius (half-width), $u_k^n = u(x_k, t_n)$, $\{(x_k, t_n) = ((k+\frac{1}{2})\Delta x, n\Delta t)$, $k \in [0, K)$, $n \in [0, N)\}$. The values of $\alpha_{\pm l}$ for specific r may be readily computed manually or generated with scripts [19].

2. Stepwise Algorithm

By the word "stepwise" we denote the most common way of stencil implementation on GPU [32]. The data is localized in the global device memory. This algorithm is the most intuitive

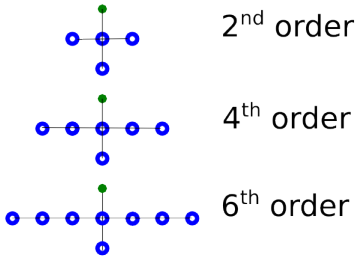


Figure 1. Cross stencils for $r = 1, 2, 3$: 3 time layers, $2r + 1$ points in space on the middle layer; the blue dots are read and the green dot is updated in the stencil computation

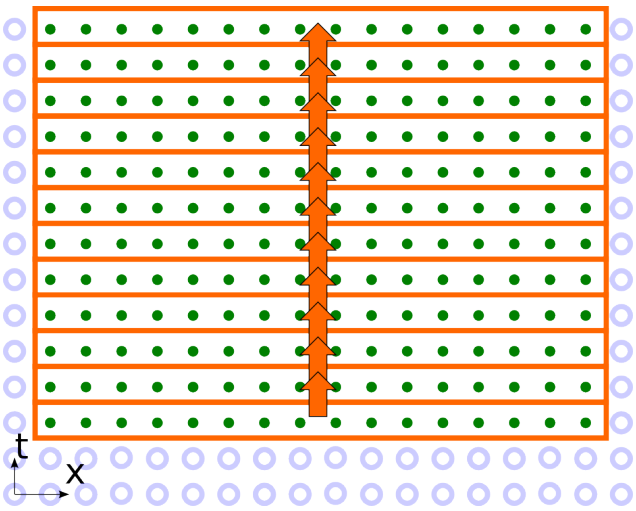


Figure 2. Stepwise algorithm

to implement with the tools available in CUDA [15]. However, even in this case care must be taken to get close to the optimal performance.

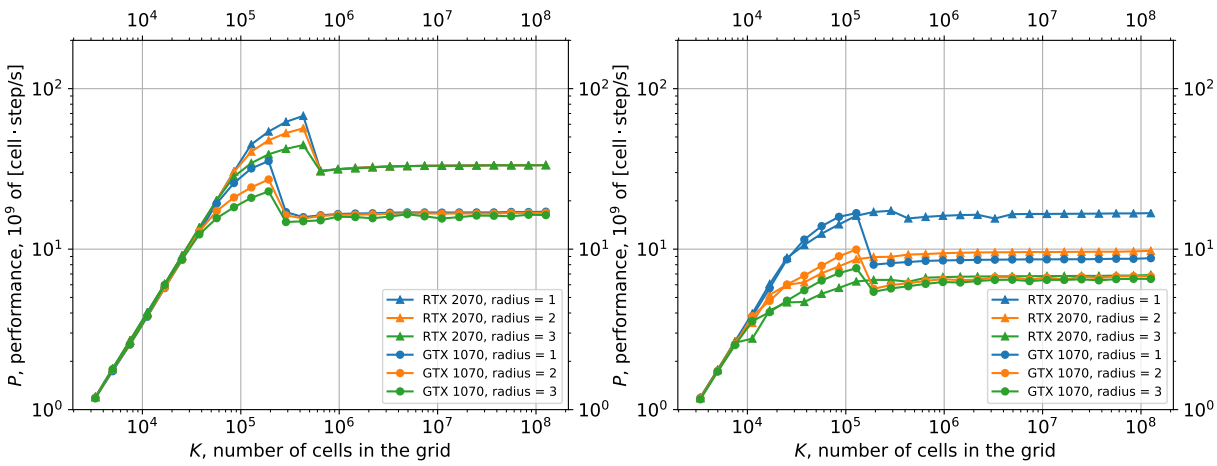
The two time layers u^n and u^{n-1} are stored in the global memory. The computation kernel computes one update for each cell according to the stencil (1). Each thread is assigned to one cell [32]. In total, `K/threads` CUDA-blocks are executed, where `threads` is the number of threads per block (usually `1024` or `512` or `256`). This CUDA-kernel is executed N times in a CPU loop.

The stepwise algorithm is illustrated in terms of the problem dependency graph in Fig. 2. One point represents one u_k^n computation. Inside the outlined areas there are no data dependencies. The domain size in x is limited by the size of the device memory, which means that about 10^9 cells may be stored. There is a data dependency between adjacent outlined areas, as shown by the arrow. After a CUDA-kernel computes one such area and exits, the data is synchronized, and the next loop iteration is started.

2.1. Performance Testing

The dependency of the performance of the stepwise kernel on the problem size has been tested. The number of cell updates per second ($\frac{\text{cell}\cdot\text{step}}{\text{s}}$) is chosen as a unit for performance P. Thus, $P = \frac{K \cdot N}{time}$ where $time$ is the time per program run in seconds.

In Fig. 3 this dependency is shown for $r = 1$, 2, 3 and for the two GPU: GTX 1070 (Pascal) and RTX 2070 (Turing). We have tested single precision (32 bit) and double precision (64 bit). Note, that target applications for desktop GPU require only single precision floating point operations, since its double precision performance is much lower.



a) Single precision b) Double precision

Figure 3. The stepwise algorithm performance dependence on problem size

At the start the linear increase is observed, due to low occupancy [1] of high parallel GPU device. The number of threads that may be started on an Streaming Multiprocessor (SM) depends on the required registers number (up to 64 thousand 32 bit registers per SM) and the shared memory size per thread (up to 48–64KB). The stepwise kernel does not use the shared memory explicitly and uses no more than 32 registers per thread. Thus, each SM may process up to 2048 cells simultaneously. There are 15 SM on GTX 1070 and 36 SM on RTX 2070. Therefore, 30 thousand and 72 thousand cells are required to utilize all resources of GTX1070 and RTX2070 correspondingly. This is in accordance with the cell number K value at which the performance P stops the linear increase. It gradually saturates, then falls to the constant $P = P_{sw} \sim 10^{10}\ \frac{\text{cell}\cdot\text{step}}{\text{s}}$.

We note that in the single-precision computation the performance does not depend on the stencil radius. This is an evidence that the current implementation is memory-bound, since the operation count increases with the stencil radius. For double precision a slight variation in performance for different r is observed. In that case, the problem is closer to compute-bound domain since the peak computing performance for double precision is much lower.

The peak at $K = 10^5$ is clearly seen in the single-precision performance, which is higher than P_{sw}. It is explained by the use of the L2 cache. Since the problem is memory-bound, P is determined by the global memory bandwidth. When the data may be localized in the L2 cache (2MB for GTX 1070, 4MB for RTX 2070), L2 memory bandwidth determines the performance. It is 1.5-2 times higher. This size corresponds to $K = 2.5 \cdot 10^5$ and $K = 5 \cdot 10^5$ cells with single precision, which is exactly the location of the end of the peak and transition to P_{sw}.

2.2. Roofline Model

The Roofline [21] is a visual performance model that provides an estimate of the performance limit from the two fundamental ceilings: peak computing performance and memory bandwidth:

$$\Pi_{alg} \leq \min(\Pi_{peak}, \Theta \cdot I_{alg}). \tag{2}$$

Here Π_{alg} is the algorithm achievable performance, and Π_{peak} is the theoretical peak computing performance in the elementary operations per unit of time. Θ is the memory bandwidth.

$I_{alg} = \frac{O_{alg}}{D_{alg}}$ is the arithmetic intensity of the algorithm, where O_{alg} is the number of operations in the algorithm and D_{alg} is the data traffic to and from memory.

Usually, Π_{peak} and Θ are taken from the device specification. However, since these values depend on the frequency which may vary in wide range, they should be calculated accurately as follows. Arithmetic instruction throughput τ is the number of instructions that may be executed per cycle. These values can be found in [3], and are determined by the GPU architecture. The number of SM μ can be obtained by CUDA Runtime API. To get the actual frequency of SM ν_{SM}, the monitoring tools (such as `nvidia-smi` or `nvidia-settings`) are run during the code execution, since the frequency may depend on many factors. Then, $\Pi_{peak} = \tau\mu\nu_{SM}$. Similarly, with the use of CUDA Runtime API the memory bus width β can be found out. With the use of monitoring tools the memory frequency ν_{mem} is measured during the code run. Thus, $\Theta = 2\beta\nu_{mem}$. The factor of 2 is explained by the Graphics Double Data Rate SDRAM (GDDR SDRAM) feature. All these values are collected in Tab. 1 and Tab. 2.

Table 1. GPU characteristics: τ is the arithmetic instruction throughput, μ is the number of SM, ν_{sm} is the clock rate, Π_{peak} is the peak performance

GPU	precision	τ $[\frac{\text{FMA}}{\text{clock}\cdot\text{SM}}]$	μ [SM]	ν_{sm} $[\frac{\text{Gclock}}{\text{s}}]$	$\Pi_{peak} = \tau\mu\nu_{sm}$ $[\frac{\text{GFMA}}{\text{s}}]$
GTX 1070	single	128	15	1.860	3571
GTX 1070	double	4	15	1.860	111.6
RTX 2070	single	64	36	1.875	4320
RTX 2070	double	2	36	1.875	135.0

Table 2. GPU characteristics: β is the memory bus width, ν_{mem} is the memory clock rate, Θ is the memory bandwidth

GPU	β $[\frac{\text{B}}{\text{clock}}]$	ν_{mem} $[\frac{\text{Gclock}}{\text{s}}]$	$\Theta = 2\beta\nu_{mem}$ $[\frac{\text{GB}}{\text{s}}]$
GTX 1070	32	3.802	243.3
RTX 2070	32	6.801	435.3

The unit for measurement of the performance Π is one Fused Multiply Add (FMA) operation, which may be one or two floating point operations (FLOP), while the symbol P is used for performance in cell updates. The compiler might optimize all operations to FMA. However, we have decided to force compiler to use FMA operations whenever possible via intrinsic (built-in) functions.

For one cell update $O_{alg} = (2r+1)\ \frac{\text{FMA}}{\text{cell}\cdot\text{step}}$ operations are required. The operations are grouped manually into FMA as follows:

$$u_k^{n+1} = \underbrace{\alpha_{\pm r}\overbrace{(u_{k+r}^n + u_{k-r}^n)}^{\text{FMA}} + \ldots + \underbrace{\alpha_{\pm 1}\overbrace{(u_{k+1}^n + u_{k-1}^n)}^{\text{FMA}} + \overbrace{\alpha_0 u_k^n - u_k^{n-1}}^{\text{FMA}}}_{\text{FMA}}}_{\text{FMA}}.$$

To compute u_k^{n+1}, $(2r+1)$ values are loaded from the u^n layer, one value is loaded from the u^{n-1} layer, and one value is saved: $(2r+3)$ values total. However, since the update is stepwise, the neighboring values that are updated by other threads may be stored in the L2 cache. Actually, each thread only loads u_k^{n-1} and u_k^n to, and writes u_k^{n+1} from the global memory. To estimate

the required number of the L2 accesses it is necessary to take into account the possibility of misaligned access to the cache lines of the neighbor cells. The number of accesses to L1 about twice more that the number of accesses to L2, and it is much faster than the global memory. Thus, the L1 exchange may be neglected in the Roofline model. Data throughput is $D_{alg} = 12\ \frac{\text{B}}{\text{cell·step}}$ for single precision and $D_{alg} = 24\ \frac{\text{B}}{\text{cell·step}}$ for double precision. The arithmetic intensity is

$$I_{alg} = \frac{O_{alg}}{D_{alg}} = \begin{cases} \frac{2r+1}{12}\ \frac{\text{FMA}}{\text{B}}, & \text{for single precision} \\ \frac{2r+1}{24}\ \frac{\text{FMA}}{\text{B}}, & \text{for double precision} \end{cases}. \tag{3}$$

The performance $\Pi_{alg}\ [\frac{\text{GFMA}}{\text{s}}]$ is computed from the performance $P_{sw}[\frac{\text{cell·step}}{\text{s}}]$, that was obtained as a saturated performance for large K in the performance tests, and $O_{alg}\ [\frac{\text{FMA}}{\text{cell·step}}]$ as

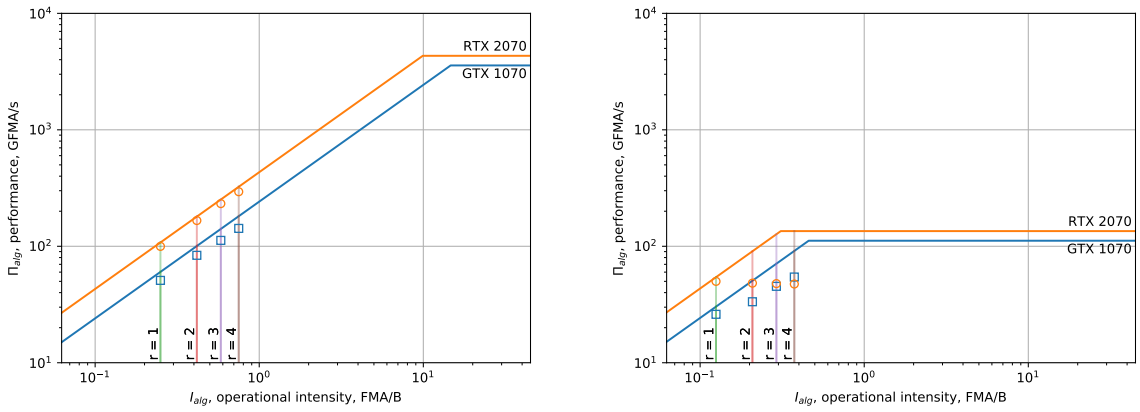
$$\Pi_{alg} = P_{sw} \cdot O_{alg}. \tag{4}$$

The Roofline graph for our implementation of the stepwise algorithm with data localization in the global memory is plotted in Fig. 4.

For single precision, our results fall into the memory-bound domain. The performance is limited by the global memory bandwidth. The overhead is observed to be negligible, as the result is close to the ceiling. With the increase of the stencil radius r the performance increases, due to the increase in the arithmetic intensity. However, peak performance to memory bandwidth ratio is about 100 times more then optimal ratio for stencil calculations with stepwise algorithm.

The double-precision peak performance on the chosen GPU is 32 times lower than single-precision peak ones, so performance to memory bandwidth ratio is closer to operation intensity. At $r = 1$ the result falls into the memory-bound area, and as the arithmetic intensity increases, the result comes closer to the compute-bound domain, and reaches it at $r = 4$. The performance is 40–50% of Π_{peak} and does not increase in the compute-bound domain.

It is likely that if a Tesla architecture GPU is used, the Roofline would look similar for single and double precision, since that architecture is better suited for the general purpose computing. This means that for the consumer-level GPU even the stepwise algorithm is enough to reach the compute-bound domain at double precision, since the double precision performance is very low. Hereafter, only single precision implementations are considered.



a) Single precision b) Double precision

Figure 4. Roofline model for the stepwise algorithm implementation. The markers show the highest performance result obtained with our code

3. Recursive Domain Decomposition

For single-precision, to move the problem closer to the compute-bound domain, the arithmetic intensity should be increased. For this purpose, we propose a Recursive Domain Decomposition (RDD) algorithm to localize the cell data in the SM registers. The classical Domain Decomposition (Fig. 5) divides the computing domain into equal domains, and each domain is assigned to its processor element. The local memory of that processor element stores u^n_{k-1} and u^n_k layer data for its domain, and the element computes cell updates for them layer by layer. Each step it exchanges data with the neighboring elements.

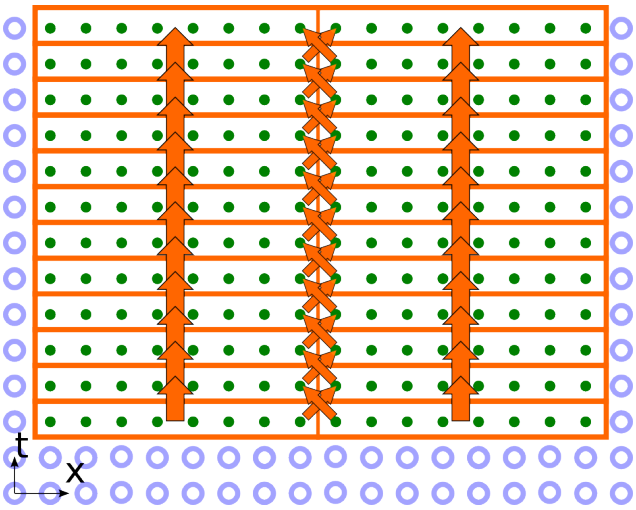


Figure 5. Domain decomposition Algorithm

However, one device contains at least 2 levels of parallelism, SM and CUDA threads. Thus, we choose to implement 2-level RDD (Fig. 6). This approach is close to the tiling (blocking) methods, such as [9], since the performance gain is achieved by the use of the faster memory for each process.

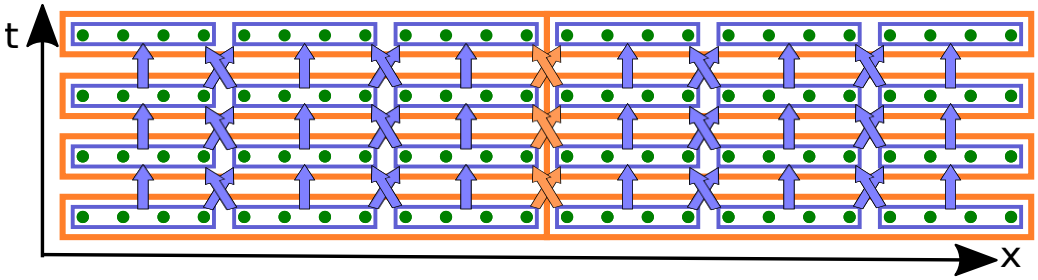


Figure 6. Two-level Recursive Domain Decomposition (RDD) Algorithm. Domains of the first level are outlined in blue, domains of the second level are outlined in orange

Each SM is assigned a domain that corresponds to the size of its register file. In turn, inside these domains thread-level decomposition occurs. Each thread is assigned with not one, like in the stepwise algorithm, but a localized group (`grp`) of cells. Each SM contains 64 thousand 32 bit registers. In the ideal case, two registers per cell are required (u^n_k and u^n_{k-1}). Thus, up to 32 thousand cells may be stored and processed per SM. Including some possible overhead, the actual number may be estimated as $20 \div 25$ thousand cells per SM. That is, $300 \div 375$ thousand cells total per 15 SM of GTX 1070, $720 \div 900$ thousand cells total per 36 SM of RTX 2070. This is more than enough for many 1D problems.

3.1. Synchronization

For the data exchange between SMs the L2 cache is used, and for the data exchange between threads the shared memory is used. The inter-block synchronization was an open issue up to the release of CUDA 9. Since CUDA 9 for devices with Compute Capability 6.1 or higher, CUDA Cooperative Groups (CG) is an universal API for synchronization [15]. It may be applied to a group of threads of arbitrary size, from one warp to all threads of several devices, situated on one node. This is a barrier synchronization, that is, a chosen group of threads is synchronized altogether. This mechanism seems convenient for our purposes, and we have applied it to our code.

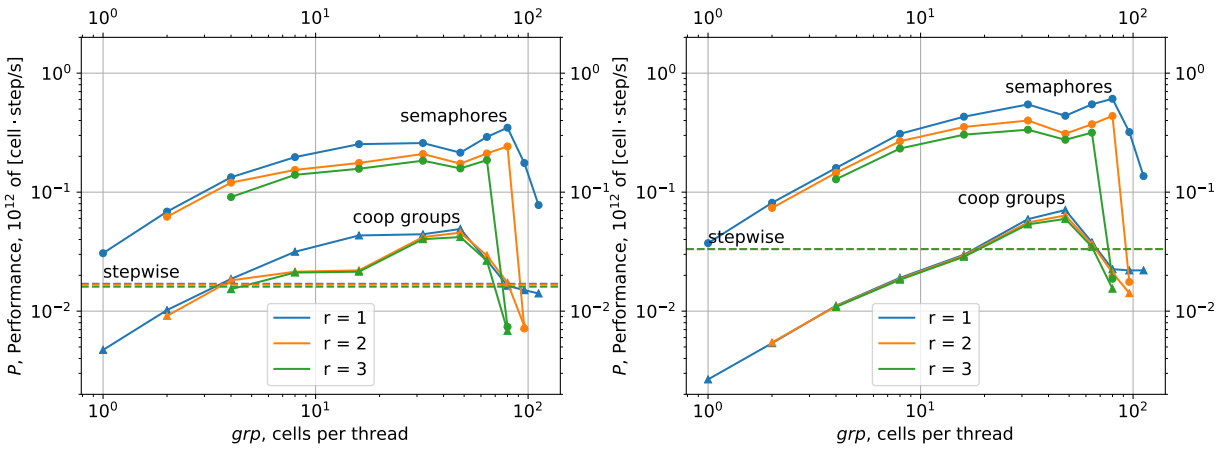
Along with it, we choose to manually implement and test the performance gain of the local block synchronization with a semaphore, the classic synchronization primitive. The semaphore limits the number of parallel processes that can read and write the shared data. In this case two neighboring blocks work with the same data: one reads and the other writes. The pointer array stores the semaphores, assigned to the data section which is subject to the racing condition. The length of the array is equal to twice the number of blocks, since each domain has two regions, which are accessed by other blocks: at the start and at the end. The semaphore data type is integer and it has two states: `READABLE (0)` and `WRITEABLE (1)`. Before reading the data chunk from the other block, the corresponding semaphore is checked to be `READABLE`, using `while` loop terminated by a semicolon. After the read from memory, the value `WRITABLE` is written into the semaphore, since no other block requires this data. The data can now be overwritten by the adjacent block.

CUDA CG synchronization appears to be inefficient for our problem (see the comparison in Fig. 7). It is the possible reason that the synchronization is called for the whole grid of blocks at once, and all SMs are synchronized. This is superfluous for ensuring the correctness of the data read. Actually, it is sufficient to wait only for the two blocks (in 1D) to complete the work up to this point. If the synchronization of one block with only its neighbors was implemented in CG, it would possibly lead to the acceleration of the memory access.

3.2. Performance Testing

The RDD algorithm is defined by the three parameters: the number of cells per thread `grp`, the number of `threads` in a block and the number of `blocks`. The number of the domains of the first level of the decomposition is equal to the number of threads, the number of the domains of the second level is equal to the number of blocks. The latter is equal to or divisible by the number of SM in GPU. In the tests we take the maximum possible number of cells, thus $\texttt{grp}\cdot\texttt{threads}\cdot\texttt{blocks} \approx const$, any two of the parameters may be varied. We have tested various `threads` and `blocks` parameters. "Heavy" blocks and threads, that is, the blocks that use so much shared memory and register space that it may be assigned to an SM only one-by-one, have proven to be more efficient. We fix the number of threads per block to 256, since it is preferable to use all of the $256^2 = 65536$ registers, and the CUDA compiler does not allow more than 256 registers per thread.

In Fig. 7 the dependency of `P` on `grp` is shown with $\texttt{threads} = 256$ and maximum possible `blocks`. The dashed line in the saturated stepwise performance is P_{sw}. The two groups of lines correspond to the two types of implemented block synchronization approaches. The cooperative groups approach seems to be not efficient enough. Indeed, with this type of synchronization, the



a) GTX 1070 b) RTX 2070

Figure 7. Performance dependency on the number of cells per thread `grp` parameter in the RDD algorithm

performance hardly exceeds the performance of the stepwise algorithm. In the following we use only the manual synchronization method with semaphores.

Low `grp` provide less performance. The performance grows linearly up to `grp = 32`. For `grp` in the $32 \div 80$ range the performance shows little variation. It shows that the performance of the code with $grp = 32$ and 2 blocks per SM and the performance of the code with $grp = 64$ and 1 block per SM are close. As `grp` exceeds some peak value, 64 or 80, the abrupt fall of performance is seen. At this point, the data could not be localized in the registers, and register spilling [3] occurs.

In total, the obtained performance is $P_{RDD} \sim (300 \div 600) \cdot 10^9$, depending on the stencil radius r.

3.3. Roofline Model

Since the two memory levels are engaged in the RDD algorithm, the two bandwidths are to be considered. Thus, the Roofline is first defined by

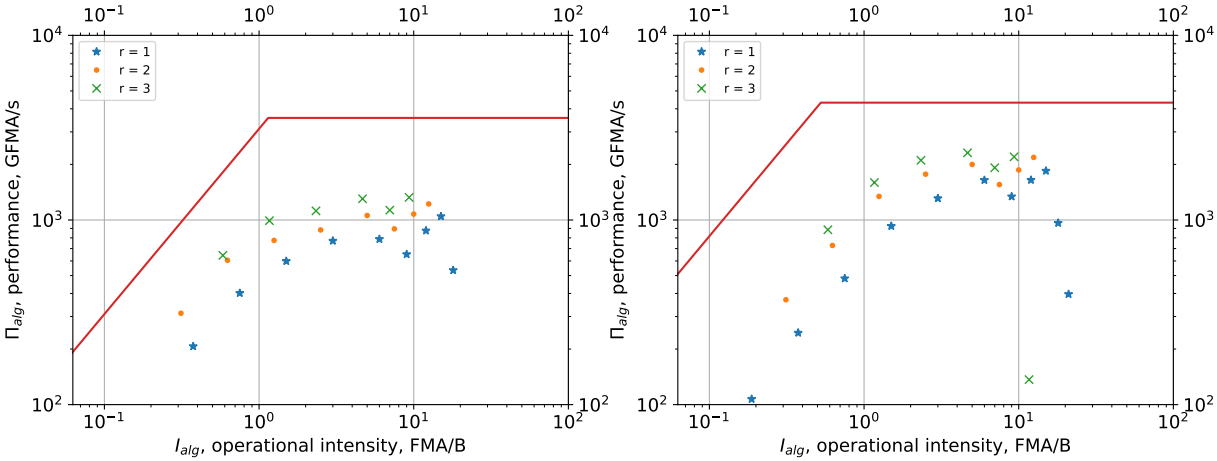
$$\Pi_{alg} \leq \min(\Pi_{peak}, \Theta_{L2} \cdot I_{alg,L2}, \Theta_{sh} \cdot I_{alg,sh}). \tag{5}$$

Here the denominations are as in section 2.2, and the extra subscript denotes the type of memory (L2 cache or shared). As before, $O_{alg} = (2r+1)\,\frac{\text{FMA}}{\text{cell}\cdot\text{step}}$. The data throughput is estimated as follows. In each block, there are $\texttt{grp}\cdot\texttt{threads}$ cells. Each block exchanges $4r$ values of 4B size. Thus, assuming $\texttt{grp} = 64$ and $\texttt{threads} = 256$,

$$D_{alg,L2} = \frac{4r}{\texttt{grp}\cdot\texttt{threads}} \cdot 4\frac{\text{B}}{\text{cell}\cdot\text{step}} \sim \frac{r}{1000}\frac{\text{B}}{\text{cell}\cdot\text{step}}.$$

Similarly, each thread has `grp` cells and exchanges $4r$ values,

$$D_{alg,sh} = \frac{4r}{\texttt{grp}} \cdot 4\frac{\text{B}}{\text{cell}\cdot\text{step}} \sim \frac{r}{16}\frac{\text{B}}{\text{cell}\cdot\text{step}}.$$



a) GTX 1070 b) RTX 2070

Figure 8. The Roofline model against shared memory bandwidth for the RDD algorithm

The operational intensity is

$$I_{alg} = \frac{O_{alg}}{D_{alg}} \sim \begin{cases} 1000 \cdot \frac{2r+1}{r} \ \frac{\text{FMA}}{\text{B}}, \text{ inter-block via L2 cache} \\ 16 \cdot \frac{2r+1}{r} \ \frac{\text{FMA}}{\text{B}}, \text{ inter-thread via shared memory} \end{cases}.$$

Table 3. GPU characteristics: α is the shared memory efficiency, μ is the number of SM, φ is the number of shared memory banks per SM, β is the bank width, ν_{gr} is the graphics clock rate, Θ_{sh} is the shared memory bandwidth

GPU	α	μ [SM]	φ $[\frac{\text{bank}}{\text{SM}}]$	β $[\frac{\text{B}}{\text{clock}\cdot\text{bank}}]$	ν_{gr} $[\frac{\text{Gclock}}{\text{s}}]$	$\Theta_{sh} = \alpha\mu\varphi\beta\nu_{gr}$ $[\frac{\text{GB}}{\text{s}}]$
GTX 1070	0.85	15	32	4	1.911	3118
RTX 2070	0.85	36	32	4	2.100	8225

The shared memory bandwidth may be calculated as $\Theta_{sh} = \alpha\mu\varphi\beta\nu_{gr}$, all these parameters and their descriptions are gathered in Tab. 3. The L2 cache bandwidth may be estimated as $\Theta_{L2} = (3 \div 5)\Theta$ [2], where Θ is taken from Tab. 2.

Now, it is easy to see that $\Theta_{sh} \cdot I_{alg,sh} \ll \Theta_{L2} \cdot I_{alg,L2}$ on actual `threads` values, therefore the inequality (5) can be reduced to

$$\Pi_{alg} \leq \min(\Pi_{peak}, \Theta_{sh} \cdot I_{alg,sh}). \tag{6}$$

The Roofline model (6) is plotted in Fig. 8. The markers show the performance for different values of `grp`. The color of a marker signifies the value of r.

The resulting performance for highest points is about 50% of the theoretical peak value. This result was obtained by localization of data in the registers and "heavy" blocks and threads ($grp \gg 1$)

Such efficiency is enough for many applications. However, in the Roofline estimate (Fig. 8) we see that another limitation acts as a bottleneck. Thus, we propose the following algorithm.

4. Recursive Domain Decomposition with Halo

Further increase in performance seems to be impeded by the high latency of L2 cache. This fact prevents the complete use of the L2 memory bandwidth. This may be mitigated by the introduction of a halo of redundant compute [20, 30].

The key idea of halo is simple: instead of the exchange of D data each step, the exchange of $\sim H \cdot D$ data each H steps takes place (Fig. 9), that is a kind of temporal tiling. Since the data is required in each step for correct simulation, the domains are overlapped.

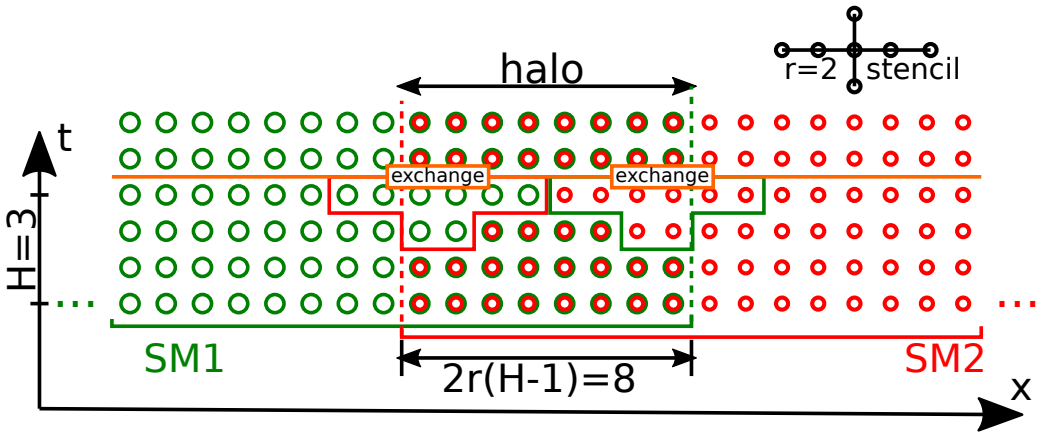


Figure 9. RDDHalo algorithms on the dependency graph. The cells that are computed correctly and stored on the two adjacent SM are shown in two colors. The data that should be sent in the synchronization is outlined in colored boxes

In the overlapped domain (halo) on the interface of two SMs, the cells are updated on both SMs (Fig. 9). At start, both SMs store the correct values of all cells, that are assigned to be updated on them, and r more values at each side that are required for the update. After the first update, the data is not enough, so r cells on each side of the domain are incorrect after the second update. The number of correct values is decreased in each step between the synchronization events each H time steps. Then, the data exchange takes place to actualize both domains. The size of the halo is $2r(H-1)$. The data to be exchanged are the cells of the half of the halo, and the r more values that are necessary for the stencil computation on the next step, $\frac{2r(H-1)}{2} + r$ total. Since the stencil requires two time layers, $\frac{2r(H-1)}{2} - r$ more values are required from the previous time step. The data required for the exchange from the two time steps is outlined by the `exchange` box in Fig. 9.

Thus, we choose to improve the RDD algorithm to the RDD with Halo (RDDhalo) algorithm by introducing the overlapped region in the inter-thread and the inter-block interfaces. Two important remarks should be made here. First of all, the overhead for redundant computation appears in comparison to the RDD, since some cells are computed twice. However, the size of the halo ($2r(H-1)$) is smaller than the size of the domain ($grp \cdot threads \sim 15 \div 20 \cdot 10^3$) by several orders of magnitude. With the increase in H the overhead may become significant, but the goal of hiding the latency by decreasing the number of synchronization events is achieved much earlier. Second, while the redundant computations may be skipped, it is better to perform these computations nonetheless. This is due to the SIMT (Single Instruction, Multiple Thread) architecture of GPU, which dictates the homogeneity of the thread computing. Thus, the conditional statements are not used in the implementation, and the incorrect cells continue to be updated until the synchronization event.

Although the halo was implemented both for inter-thread (H_b) and inter-block (H_g) interfaces, only the inter-block halo has a significant influence on the performance. The shared memory latency is low enough so that $H_b = 1$ (no halo) or $H_b = 2$ is enough to completely conceal it.

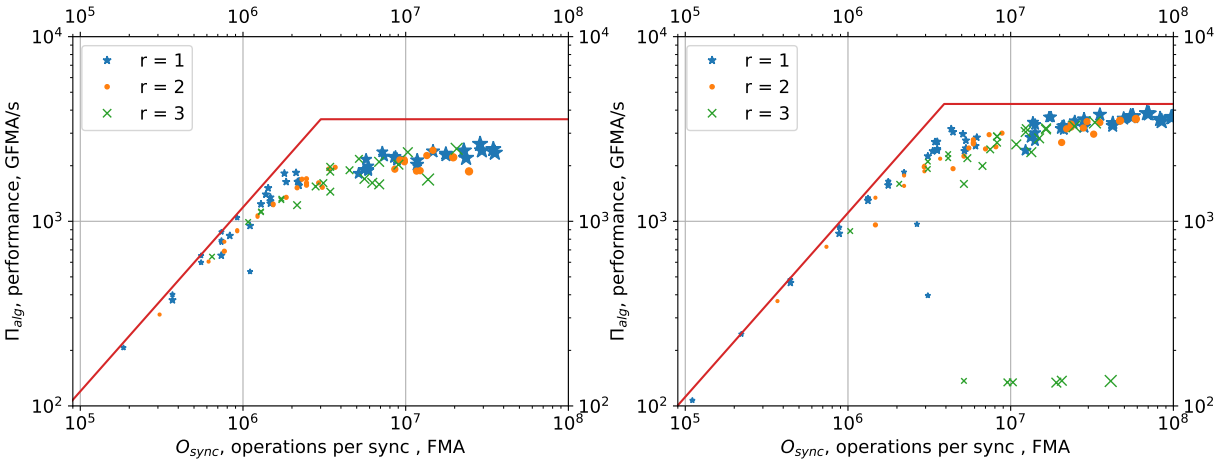
4.1. Roofline Model

The operational intensity is twice lower than in the RDD algorithm, since two time layers are exchanged each time. Nevertheless, it is still high enough for the problem to be compute-bound.

On the other hand, the latency overhead limits the performance. We may write

$$\Pi_{alg} \leq \min(\Pi_{peak}, \Theta_{L2} I_{alg,L2}, \Theta_{sh} I_{alg,sh}, O_{\text{sync}}/\Lambda_{\text{sync}}), \tag{7}$$

where $O_{\text{sync}} = KHO_{alg}$ is the operation count between synchronizations, K is the number of cells, $\Lambda_{\text{sync}} = \min\limits_{r,grp} t_{step}$ is the inter-block synchronization time, where t_{step} is the elapsed time to perform one step of RDD algorithm ($H_g = 1$). $\Lambda_{\text{sync}} = 0.84\mu$s and $\Lambda_{\text{sync}} = 0.90\mu$s for GTX 1070 and RTX 2070, respectively. The Π_{peak} and $O_{\text{sync}}/\Lambda_{\text{sync}}$ limits are visualized as a Roofline in Fig. 10. The markers show the performance of our implementation for different values of `grp`, H_g and H_b. The graph confirms the attention to the latency overhead, and the fact that it is overcome with the introduction of halo.



a) GTX 1070 b) RTX 2070

Figure 10. Latency limitation. The markers show the results of the test runs with different halo and `grp` parameters. Larger marker corresponds to larger H_g

The highest performance we achieved and the parameters that were used are presented in Tab. 4.

Conclusion

In this work, we have brought to attention several insights on the high performance stencil code GPU implementation. We have carried out performance limits analysis with the Roofline model. In this paper we have detailed the method for acquisition of both hardware (memory bandwidth of all levels) and algorithmic (operational intensity) parameters. Since the results that were obtained by our code match the obtained performance limits precisely, we can conclude

Table 4. The highest performance results obtained with RDDhalo

GPU	r	`grp`	H_b	H_g	$P, \frac{\text{Gcell·step}}{\text{s}}$	P/P_{peak}, %
GTX 1070	1	80	2	32	873	76
GTX 1070	2	48	1	8	484	69
GTX 1070	3	48	1	8	352	70
RTX 2070	1	80	2	32	1281	92
RTX 2070	2	80	1	16	716	84
RTX 2070	3	64	1	8	490	80

that the implementation has minimal overhead and that the Roofline estimation was correct. We have found a way to get a performance of more than 90% of the compute-bound limitation, and more than 80% for larger stencil radius.

According to our previous experience, in the practice of implementation of codes in scientific computing, the omnipresent issue is to evaluate the progress on performance optimization. That is, whether the code has reached the performance limit, or it may be further improved by increasing the operational intensity, utilizing better programming tools, reducing overhead. The impact of the current paper is the evidence that the stencil codes performance may be driven close to the peak if the computations are localized in the highest level of the memory hierarchy of GPU, namely in the register file. In the stepwise implementation, the obtained efficiency value is $\sim 10^{10} \frac{\text{cell·step}}{\text{s}}$, and in the RDDhalo algorithm, it is $\sim 10^{12} \frac{\text{cell·step}}{\text{s}}$. Both correspond closely to the Roofline estimate. The RDDhalo performance is compute-bound and reaches 92% of the peak computing performance.

We assume that it is the computing performance peak limit in any further stencil codes implementations. Such codes are relevant, for example, in electromagnetic wave simulation with the FDTD method, elastic wave simulation with the Levander scheme. More complex multi-level numerical schemes often use the cross stencil as one of the computation stages as well [17]. As is, the wave equation simulation is used both in optics and in seismic codes. In case the purpose of the simulation is the solution of inverse problem, such as image reconstruction in seismic exploration, double precision is excessive and single precision will suffice.

Any other stencil code, including the 3D extension of the current scheme, is more complex than the one used in this work. So the obtained performance efficiency value is unlikely to be surpassed on the current hardware. The data size in the problem fits the register file, which is comparatively small. However, the common trend in newer GPU is the increase of the register file space. It is 3.75 MB on Kepler, 6 MB on Maxwell, 14 MB on Pascal, 20 MB on Volta [16].

For 3D problems, our approach may be used in temporal tiling algorithms. As an example, in the wavefront tiling, one wavefront slice may fit in the register file. In case it is large enough, the memory transactions with the global memory may be concealed, and the performance in the tile would determine the performance efficiency, save for the newly introduced overhead. We intend to use the underlying algorithm in our projects on developing 3D wave simulation codes with LRnLA algorithms [26].

References

1. De Donno, D., Esposito, A., Tarricone, L., Catarinucci, L.: Introduction to GPU computing and CUDA programming: A case study on FDTD [EM programmer's notebook]. IEEE Antennas and Propagation Magazine 52(3), 116–122 (2010), DOI: 10.1109/MAP.2010.5586593

2. Jia, Z., Maggioni, M., Smith, J., Scarpazza, D.P.: Dissecting the NVidia Turing T4 GPU architecture via microbenchmarking. arXiv: 1903.07486 (2019)

3. Jia, Z., Maggioni, M., Staiger, B., Scarpazza, D.P.: Dissecting the NVidia Volta GPU architecture via microbenchmarking. arXiv: 1804.06826 (2018)

4. Hou, K., Wang, H., Feng, W.c.: GPU-unicache: Automatic code generation of spatial blocking for stencils on GPUs. In: Proceedings of the Computing Frontiers Conference, May 15–17, 2017, Siena, Italy. pp. 107–116. ACM, New York, NY, USA (2017), DOI: 10.1145/3075564.3075583

5. Endo, T.: Applying recursive temporal blocking for stencil computations to deeper memory hierarchy. In: 2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Hakodate, Japan, August 28-31, 2018. pp. 19–24. IEEE (2018), DOI: 10.1109/NVMSA.2018.00016

6. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: ACM Sigplan Notices. vol. 26, pp. 30–44. ACM (1991)

7. Barba, L.A., Yokota, R.: How will the fast multipole method fare in the exascale era. SIAM News 46(6), 1–3 (2013)

8. Yount, C., Duran, A.: Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling. In: 2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 65–75. IEEE, Salt Lake, UT, USA (Nov 2016), DOI: 10.1109/PMBS.2016.012

9. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Austin, Texas November 15-21, 2008. pp. 4:1–4:12. IEEE Press, Piscataway, NJ, USA (2008), DOI: 10.1109/SC.2008.5222004

10. Rawat, P.S.: Optimization of stencil computations on GPUs. Ph.D. thesis, The Ohio State University (2018)

11. Rivera, G., Chau-Wen Tseng: Tiling optimizations for 3D scientific computations. In: ACM/IEEE SC 2000 Conference (SC'00), November 04-10, 2000, Dallas, TX, USA. p. 32. IEEE (2000), DOI: 10.1109/SC.2000.10015

12. Prokop, H.: Cache-oblivious algorithms. Ph.D. thesis, Massachusetts Institute of Technology (1999)

13. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, Louisiana, November 13-19, 2010. pp. 1–13. IEEE Computer Society (2010), DOI: 10.1109/SC.2010.2

14. Fukaya, T., Iwashita, T.: Time-space tiling with tile-level parallelism for the 3D FDTD method. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, Chiyoda, Tokyo, Japan, January 28-31, 2018. pp. 116–126. HPC Asia 2018, ACM, New York, NY, USA (2018), DOI: 10.1145/3149457.3149478

15. NVIDIA Corporation: CUDA C programming guide. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` (2019), pG-02829-001_v10.1, accessed: 2019-06-18

16. NVIDIA Corporation: NVIDIA Tesla V100 GPU architecture. the worlds most advanced data center GPU. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf` (2017), wP-08608-001_v1.1, accessed: 2019-06-18

17. Korneev, B., Levchenko, V.: Detailed numerical simulation of shock-body interaction in 3D multicomponent flow using the RKDG numerical method and DiamondTorre GPU algorithm of implementation. In: Journal of Physics: Conference Series. vol. 681, p. 012046. IOP Publishing (2016), DOI: 10.1088/1742-6596/681/1/012046

18. Zakirov, A., Levchenko, V., Perepelkina, A., Zempo, Y.: High performance FDTD algorithm for GPGPU supercomputers. In: Journal of Physics: Conference Series. vol. 759, p. 012100. IOP Publishing (2016), DOI: 10.1088/1742-6596/759/1/012100

19. Fornberg, B.: Generation of finite difference formulas on arbitrarily spaced grids. Mathematics of computation 51(184), 699–706 (1988)

20. Maruyama, N., Aoki, T.: Optimizing stencil computations for NVIDIA Kepler GPUs. In: Proceedings of the 1st International Workshop on High-Performance Stencil Computations, January 21, 2014, Vienna, Austria. pp. 89–95

21. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM 52(4), 65–76 (2009), DOI: 10.1145/1498765.1498785

22. Quilleré, F., Rajopadhye, S., Wilde, D.: Generation of efficient nested loops from polyhedra. International journal of parallel programming 28(5), 469–498 (2000), DOI: 10.1023/A:1007554627716

23. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with Lift. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018, February 24-28th 2018, Vienna, Austria. pp. 100–112. ACM Press, Vienna, Austria, DOI: 10.1145/3168824

24. Phillips, E.H., Fatica, M.: Implementing the Himeno benchmark with CUDA on GPU clusters. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), April 19-23 2010, Atlanta, GA, USA. pp. 1–10. IEEE (2010), DOI: 10.1109/IPDPS.2010.5470394

25. Levchenko, V.D.: Asynchronous parallel algorithms as a way to archive effectiveness of computations. Journal of Inf. Tech. and Comp. Systems (1), 68 (2005), (in Russian)

26. Levchenko, V.D., Perepelkina, A.Y.: Locally recursive non-locally asynchronous algorithms for stencil computation. Lobachevskii Journal of Mathematics 39(4), 552–561 (2018), DOI: 10.1134/S1995080218040108

27. Muranushi, T., Makino, J.: Optimal temporal blocking for stencil computation. Procedia Computer Science 51, 1303–1312 (2015), DOI: 10.1016/j.procs.2015.05.315

28. Muranushi, T., Nishizawa, S., Tomita, H., Nitadori, K., Iwasawa, M., Maruyama, Y., Yashiro, H., Nakamura, Y., Hotta, H., Makino, J., et al.: Automatic generation of efficient codes from mathematical descriptions of stencil computation. In: Proceedings of the 5th International Workshop on Functional High-Performance Computing, Nara, Japan, September 22, 2016. pp. 17–22. ACM, New York, NY, USA, DOI: 10.1145/2975991.2975994

29. Riesinger, C., Bakhtiari, A., Schreiber, M., Neumann, P., Bungartz, H.J.: A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters. Computation 5(4), 48 (2017), DOI: 10.3390/computation5040048

30. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: Proceedings of the 26th ACM international conference on Supercomputing, San Servolo Island, Venice, Italy June 25-29, 2012. pp. 311–320. ACM, DOI: 10.1145/2304576.2304619

31. Krotkiewski, M., Dabrowski, M.: Efficient 3D stencil computations using CUDA. Parallel Computing 39(10), 533–548 (2013), DOI: 10.1016/j.parco.2013.08.002

32. Micikevicius, P.: 3D finite difference computation on GPUs using CUDA. In: Proceedings of 2nd workshop on general purpose processing on graphics processing units, Washington, D.C., USA, March 08, 2009. pp. 79–84. ACM, New York, NY, USA, DOI: 10.1145/1513895.1513905

DOI: 10.14529/jsfi190208

Distinct Element Simulation of Mechanical Properties of Hypothetical CNT Nanofabrics

Igor A. Ostanin[1,2]

A universal framework for modeling composites and fabrics of micro- and nanofibers, such as carbon nanotubes, carbon fibers and amyloid fibrils, is presented. Within this framework, fibers are represented with chains of rigid bodies, linked with elastic bonds. Elasticity of the bonds utilizes recently developed enhanced vector model formalism. The type of interactions between fibers is determined by their nature and physical length scale of the simulation. The dynamics of fibers is computed using the modification of rigid particle dynamics module of the waLBerla multiphysics framework. Our modeling system demonstrates exceptionally high parallel performance combined with the physical accuracy of the modeling. The efficiency of our technique is demonstrated with an illustrative mechanical test on a hypothetical carbon nanotube textile. In this example, the elasticity of the fibers represents the coarse-grained covalent bond within CNT surface, whereas interfiber interactions represent coarse-grained van der Waals forces between cylindrical segments of nanotubes. Numerical simulation demonstrates stability and extremal strength of a hypothetical carbon nanotube fabric.

Keywords: nanofibers, carbon nanotubes, distinct element method, parallel computing.

Introduction

Fibrillar materials, based on biological fibrils, carbon fibers, nanofibers and carbon nanotubes (CNTs) [1–4], and, in particular, textiles and fabrics made of individual fibrils or woven fibers, are of extreme interest for a number of military, aerospace, electronic and biomedical applications. However, intricate hierarchical structures of such materials, their discontinuous behavior with non-trivial inter-fiber interactions, as well as the prohibitively large sizes of representative volume elements in many cases prevent straightforward theoretical prediction of the mechanical, electrical and thermal properties of such materials. Understanding of the mesoscale behavior of these materials can be improved via numerical simulations. Atomic-level modeling techniques [5–8], namely – tight-binding and molecular dynamics methods, were proved to be efficient numerical tools for modeling individual fibrils and their interactions at the nanoscale, however, the scalability of such techniques is insufficient for modeling large numbers of long fibers, necessary for studying the mechanics of the sufficiently large specimens of fibrillar materials. In order to address this problem, a number of mesoscale models were suggested. One of them, bead-spring model, employs the idea of coarse-grained molecular dynamics [9–11], initially proposed for modeling mesoscale mechanics of proteins. In this modeling concept, a chain of point masses represents a fibril interacting via classical potentials, representing either intra-fibril elasticity, or contact interactions between the neighboring fibrils. This model, despite its evident advantages, has certain limitations in a context of modeling fibrillar materials and fabrics. The most important of them is absent torsional stiffness of fibrils leading to unrealistic behaviors of fibrillar assemblies under certain loadings. In order to solve this issue, a different discretization concept [12–19] was suggested using a representation of a thin fiber as a chain of rigid bodies, rather than point masses. Such a model allows not only bending of individual fibers, but their torsion as well. This simulation technique, known as mesoscopic distinct element method

[1]Skolkovo Institute of Science and Technology, Moscow, Russian Federation
[2]University of Twente, Enschede, Netherlands

(MDEM), established itself in the field of modeling CNTs systems as one of the most efficient mesoscopic modeling tools, both computationally efficient and physically just. The technique can be successfully used for modeling a wide class of fibers and fibrillar materials, on the scales that admit athermal description of the fiber mechanics. Until now, the remaining obstacle on the route toward applications of MDEM to large-scale modeling of fibrillar assemblies was the absence of its scalable, parallel realization. Such realization has been recently suggested in [19]. It is based on rigid particle dynamics module of the waLBerla multiphysics framework [20]. In the current work we illustrate the novel modeling approach in application to modeling nanofabrics – hypothetical textiles made of single wall carbon nanotubes (CNTs), ultimately strong nanofilaments.

1. Method

1.1. Mesoscopic Model

Our model is based on the mesoscopic distinct element method, that computes the damped dynamics of a collection of interacting classical particles with certain mass and tensor of inertia. We utilize the spherical particles with the radius r, uniformly distributed mass m and the scalar moment of inertia $I = \frac{2}{5}mr^2$. The state variables for each particle include translational positions and velocities, as well as rotational positions (in a shape of quaternions, as described in [20]) and angular velocities. The bodies change their velocities and angular velocities due to contact forces and moments arising in pair interactions, as well as external forces and moments, acting at each body. The system is evolved in time with explicit velocity Verlet time integration scheme. Two kinds of damping may be introduced in the system. The viscous damping forces, proportional to relative segment velocities, act in parallel with pair contact forces.

PFC-style *local damping* [21] acts at each body. It is introduced to damp stiff interactions and stabilize time integration in dense particle assemblies. The components of damping force F_i^α (moment M_i^α) are proportional to the corresponding components of unbalanced force F_i (moment M_i) according to:

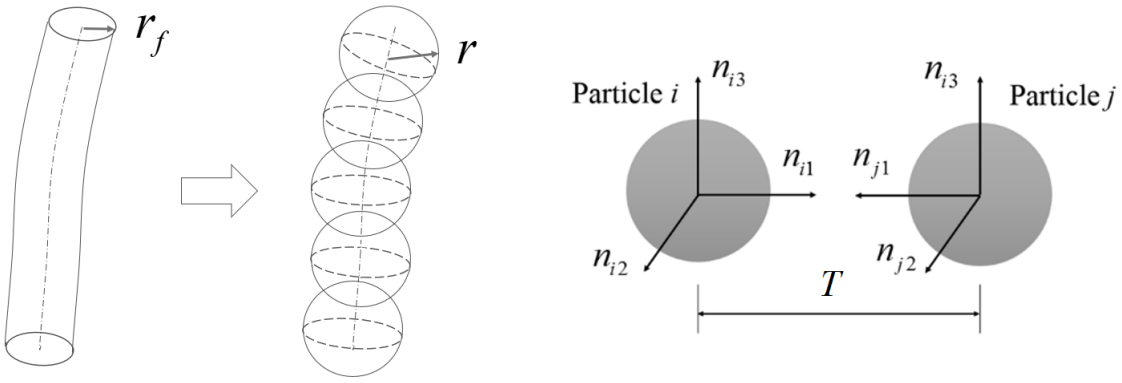
$$F_i^\alpha = -\alpha \left|F_i\right| sign(v_i), M_i^\alpha = -\alpha \left|M_i\right| sign(\omega_i). \tag{1}$$

Here v_i and ω_i are components of the translational and rotational velocity of an element, and $sign(x)$ is the sign function. In our simulations the local damping coefficient α is set to 0.4.

Within our approach, undeformed fibers are partitioned into identical segments of finite lengths $T = 2r_f$ and represented with chains of spherical rigid bodies (Fig. 1). Each spherical particle represents the inertial properties of a fiber segment - parameters m and I are equal with the mass and moment of inertia of a cylindrical segment taken with respect to the fiber axis. It follows that the spherical particle has a radius

$$r = \sqrt{2.5} r_f. \tag{2}$$

Elasticity of fibers in our model is represented with the formalism of enhanced vector model (EVM) [22, 23]. The EVM is based on a binding potential, describing the behavior of an elastic bond linking two rigid bodies. The formulation provides straightforward generalization on the case of large strains and accounts for a bending-twisting coupling. Consider two equal-sized spherical particles i and j with equilibrium separation T and equilibrium orientation described



a) Schematics of a fiber discretization b) EVM bond schematics

Figure 1. DEM discretization of CNT

in terms of orthogonal vectors n_{ik}, as depicted in Fig. 1(b) (note that for an undeformed bond $n_{i1} = -n_{j1}$, $n_{i2} = n_{j2}$, $n_{i3} = n_{j3}$). Then the EVM bond potential is given as follows:

$$U(r_{ij}, \mathbf{n}_{ik}, \mathbf{n}_{jk}) = \frac{B_1}{2}(r_{ij} - T)^2 + \frac{B_2}{2}((\mathbf{n}_{j1} - \mathbf{n}_{i1})\mathbf{r}_{ij}/r_{ij} + 2) + B_3\mathbf{n}_{i1}\mathbf{n}_{j1} - \frac{B_4}{2}(\mathbf{n}_{i2}\mathbf{n}_{j2} - \mathbf{n}_{i3}\mathbf{n}_{j3}). \quad (3)$$

Here $\mathbf{r}_{ij}$ is the radius vector connecting centers of bonded particles. The first term of the potential 3 accounts for the elastic strain energy stored due to axial tension/compression of a bond, second term is nonzero if shear deformation of the bond is present, third and fourth terms give bond's bending and torsion energy of a bond. Parameters $B_1 ... B_4$ are directly related to longitudinal, shear, bending, and torsional rigidities of a bond, according to Euler-Bernoulli beam theory (see papers [17, 22, 23] for more details):

$$\begin{aligned} B_1 &= \frac{ES}{T}, \\ B_2 &= \frac{12EJ}{T}, \\ B_3 &= -\frac{2EJ}{T} - \frac{GJ_p}{2T}, \\ B_4 &= \frac{GJ_p}{T}. \end{aligned} \quad (4)$$

Here E and G are the bond material Young's and shear moduli. Area S, moment of inertia J and polar moment of inertia J_p of a cylinder shell beam with radius r_f and thickness h are given by:

$$\begin{aligned} S &= 2\pi h r_f, \\ J &= \pi h r_f (r_f^2 + h^2/4), \\ J_p &= 2J. \end{aligned} \quad (5)$$

As an example, consider here the parameterization of our discretization scheme for single-wall CNTs. Table 1 provides segment parameters for $(10, 10)$ CNTs with diameter $2r_f = 13.56$ $\mathring{A}$ and length $T = 2r_f$. Each segment contains approximately 220 carbon atoms. Microscopically computed Young's $E = 1,029$ GPa and shear modulus $G = 459$ GPa [7] are used.

The interactions between fiber segments (*e.g.* Hertzian elastic repulsion, van der Waals (vdW) adhesion, wet surface tension, hydrogen bonds *etc.*) generally depend on the specifics

Table 1. Parameterization of the spherical particles and EVM bonds for a $(10, 10)$ CNTs. m, r, I are the mass, radius, moment of inertia of each spherical particle. B_1,B_2,B_3,B_4 are EVM stiffnesses

m	r	I	B_1	B_2	B_3	B_4
(amu)	(Å)	$(amu \times$ Å$^2)$	$(eV/$Å$^2)$	(eV)	(eV)	(eV)
$2,649$	10.72	1.218×10^5	67.59	19780	-4032	1471

Table 2. Parameterization of the fiber interaction potential 6

ε,(eV)	A	B	α	β	$k, eV/$Å^2
0.07124	0.0223	1.31	9.5	4.0	200

of a particular problem. In our example, we utilize the combination of linear elastic repulsion between fiber segments at small distances, combined with vdW adhesion at large distances. The total potential of pair interaction is given by:

$$U(r_{ij}) = \begin{cases} 4\varepsilon\left(\frac{A}{(r_{ij}/r_f-2)^\alpha} - \frac{B}{(r_{ij}/r_f-2)^\beta}\right) & r_{ij} > r_f \\ k(r_{ij}-r_f)^2 & r_{ij} \le r_f \end{cases}. \tag{6}$$

For the distances that are less than two fiber radii the potential 6 describes linear elastic repulsion. The stiffness k is fitted to ensure stable integration at a given timestep and the absence of fibers interpenetration. For the distances larger than two fiber radii, the potential 6 describes the coarse-grained potential for vdW adhesion. The calibration of a coarse-grained isotropic vdW potential for $(10, 10)$ CNTs is given in [13].

In order to capture geometric anisotropy of the cylindrical segments of fibers, we utilize simple numerical integration of the spherically-symmetric potential (6) over the length direction of segments. Three equispaced integration points along each segment's axis are employed. Table 2 provides the parameterization of the potential.

1.2. Parallel Implementation

Parallel damped dynamics simulations are based on the rigid particle dynamics module of waLBerla multiphysics framework, which is available under GPL license at (`www.walberla.net`). The parallelization is based on standard Message Passing Interface (MPI) [24] for distributed memory architectures. A complete description of the parallel algorithms and their realization [18] is beyond the scope of this paper. The simplified pipeline of our modeling framework is given in Fig. 2. The simulation starts with the generation of initial geometry of fibers, and imposition of boundary conditions. In the next step, the simulation domain is divided in a balanced manner into rectangular subdomains. These subdomains are distributed among the available MPI processes in such a way that every process is responsible for one or more subdomains. At the next stage, time integration cycles are performed on all MPI processes. The integration cycle consists of computation of pair interaction potentials, as well as the corresponding forces and moments at each contact. These forces and moments are then used to compute accelerations and angular accelerations that are then used in computing updated positions and velocities

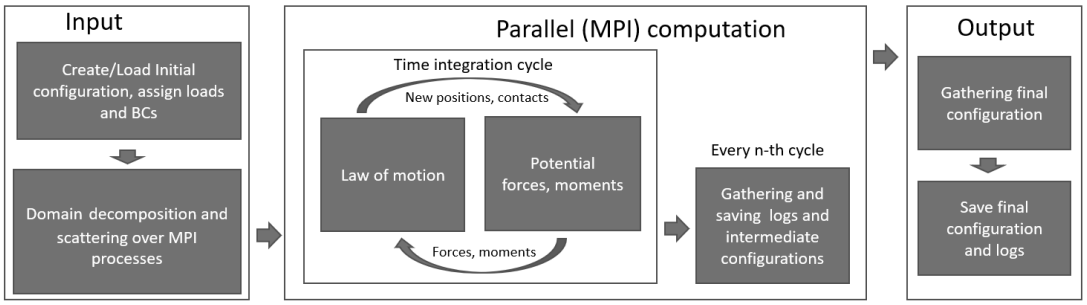


Figure 2. Simplified diagram of the modeling framework pipeline

according to velocity Verlet time integration scheme. Particle migrations across subdomain borders are accounted via MPI communications. Then the list of contacts is updated. The contact detection scheme used in our simulations is based on hierarchical hash grids [25] and adapted for potential-based interactions. For correct contact detection of particles near the borders of a subdomain so-called ghost particles are introduced. These ghost particles mirror particles which touch the subdomain but are located at a different one. This way they are available for contact detection and force calculation. The configuration and traced quantities (*e.g.* potential energy) are gathered at and saved periodically during the relaxation. The general scalability of waLBerla framework is proven up to almost half a million cores [20]. In the simulations presented below only 480 cores were utilized, which is well below the limits of linear strong scalability. Some auxiliary serial operations (*e.g.* gathering of the total potential and kinetic energy of the system) can in principle limit the efficiency of parallelization, however, they are not a necessary part of the computation cycle.

2. Numerical Results

As an example of application of our system, we consider here the relaxation and mechanical test on a hypothetical CNT textile material. Modern technologies do not yet allow to produce such textiles, however, our modeling framework allows to evaluate its properties in a mesoscopic simulation.

The simulations were performed at a computational cluster "Zhores" [26] using 20 nodes. Every node uses two Intel Xeon 6136 Gold CPUs (24 cores, 3 GHz each). The high performance cluster network has the Fat Tree topology and is built from six Mellanox SB7890 (unmanaged) and two SB7800 (managed) switches that provide 100 Gbit/s connections between the nodes.

Consider a fragment of a CNT fabric, consisting of 400 CNTs, 1.35 μm long each (2.4×10^6 model degrees of freedom), forming a square piece of a textile (Fig. 3a). Two rows of CNTs, 200 CNTs each, are intertwined in a manner of a "mosquito net" made of individual CNTs, as depicted on a lower inset of Fig. 3a. Individual CNTs in a fabric are deformed in a shape of a sine wave, as shown on the upper inset of Fig. 3a. The doubled sine magnitude L and half-period D are equal to 33.7 Å and 17.1 Å, respectively. Since CNT elongation due to sinusoidal shape is significant, CNTs in an equilibrated periodic specimen are subjected to both bending and stretching. We first apply periodic boundary conditions along x and y directions, and perform the initial relaxation of the "mosquito net" configuration. It appears that this configuration is stable, and the initial prescribed shape is very close to a local minimum. However, it is interesting

to answer the question about the stability of a *finite-sized* piece of such material. To this aim, after short initial relaxation (1000 integration cycles), we remove periodic boundary conditions, leaving the edges of CNTs force-free. After that the specimen is allowed to relax in a damped dynamics simulation to a meta-stable state. At the initial stage of the simulation, edge CNTs start to separate from the fabric, since low vdW adhesion energy can not confine elastic strain energy, which is released during separation of side CNTs. Detached side CNTs form bundles comprising about 10–20 tubes each (Fig. 3b and 3d). However, at the next stage of simulation the fabric disintegration process stops, while the relaxation slows down. CNTs do not separate from the fabric, since further separation is prevented by vdW adhesion. Similarly to the cases of other self-assembled CNT structures [14–16], CNT fabric achieves a meta-stable state, which is characterized by the balance between vdW adhesion energy and elastic strain energy. Such a balance is achieved for structure features of a certain size, characterized by the mesoscopic length scale

$$l_0 = \sqrt{\frac{EJ}{\eta}}, \tag{7}$$

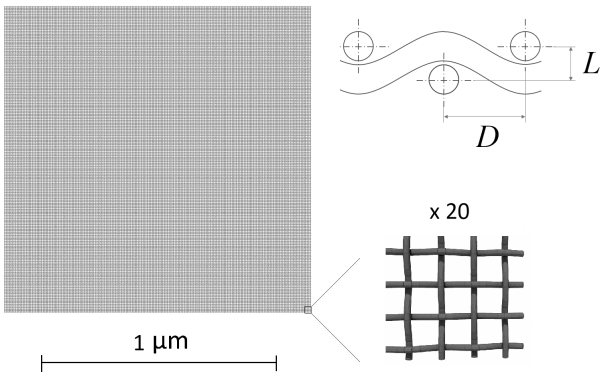
where EJ is the bending stiffness of a CNT, and η is the vdW adhesion energy per unit length. This length scale arises explicitly in the analysis of elementary self-folded configurations - rings [13], rackets [16], multiple-winding rings [15]. For an individual $(10, 10)$ CNTs, given the bending stiffness of 22350 eVÅ and the adhesion energy of 0.22 eV/Å, this length scale is equal to 0.032 μm. For bundles, comprising multiple CNTs, both bending stiffness and characteristic length scale are somewhat larger.

Figure 3c gives the dependence of the elastic strain energy stored in separate CNTs during the relaxation. As we can see, the dependence is nearly exponential and strongly indicates the convergence to a stable state. Thus, we have a numerical evidence of the stability of hypothetical CNT fabrics.

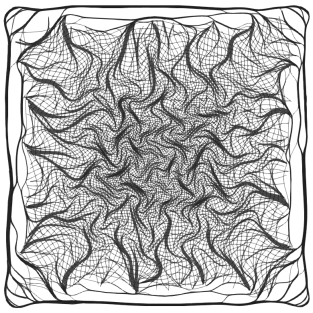
In order to demonstrate *in silico* the exceptional mechanical properties of this material, we perform a numerical simulation of a large strain displacement controlled mechanical test on a specimen of CNT fabric material. Figure 4a–c illustrates the experiment setup. Self-assembled equilibrated CNT film specimen is subdivided into three regions - two grips, marked with green and red colors, and a gage region, marked with blue. Starting from the initial moment of the simulation, grips start to move in opposite directions, stretching the gage region. Grip velocity is kept constant, providing strain rate of 2×10^8 s^{-1}, with exception for short constant acceleration period in the beginning of the simulation, necessary to avoid inertial peak at the beginning.

For this test, we introduced a simple breakage model for a CNT in assembly. An individual CNT breaks if it is stretched up to certain critical level. This critical strain has a normal distribution with the mean value ϵ_c and dispersion $\Delta\epsilon$; random distribution is introduced to qualitatively evaluate effects of finite temperature and CNT defects. In our test, $\epsilon_c = 0.3$, $\Delta\epsilon = 0.05$.

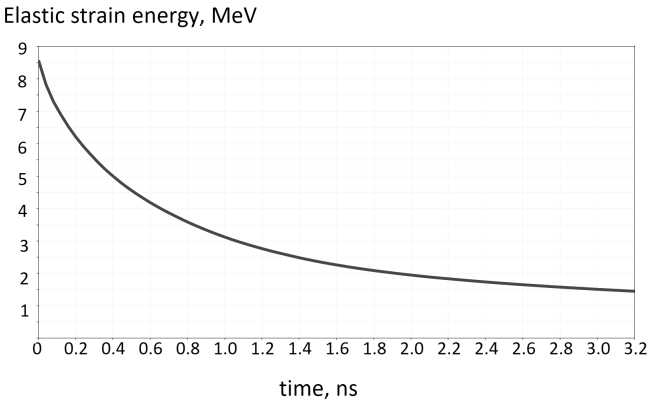
Figure 4d displays stress-strain curves during the simulation. Stress is defined in the assumption that the fabric's thickness is equal L (Fig. 3a). The initial step of the stress-strain curve is associated with dissipative response, conditioned by presence of local damping. The subsequent hardening region (strains of 2–20%) is associated with straightening of individual CNTs in fabric, oriented along the loading direction (Fig. 4b). At strains of 20–25% we can see the elastic response conditioned by stretching of individual CNTs. Region of strains of 25–50% features complex failure with correlated breakage of separate CNTs (Fig. 4c), first at the edges



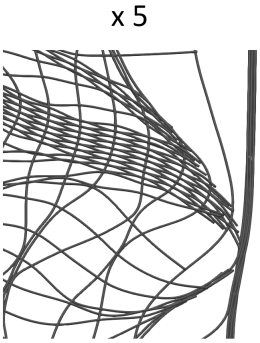
a) initial configuration



b) Final relaxed configuration



c) Elastic strain energy stored in CNTs as a function of time



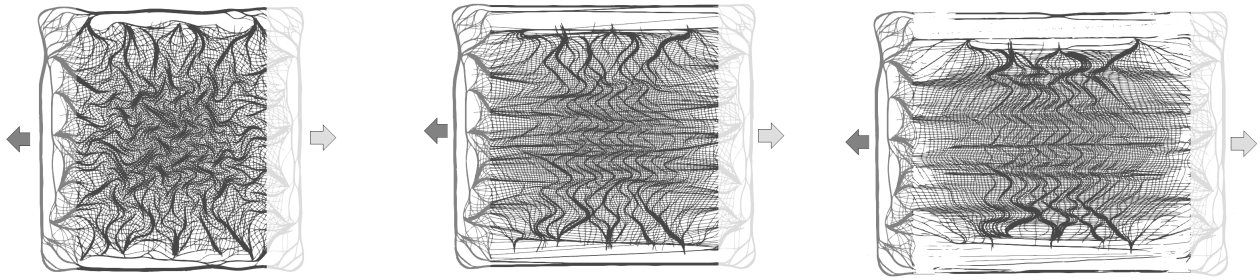
d) Magnified piece an equilibrated configuration

Figure 3. Relaxation of the CNT fabric specimen

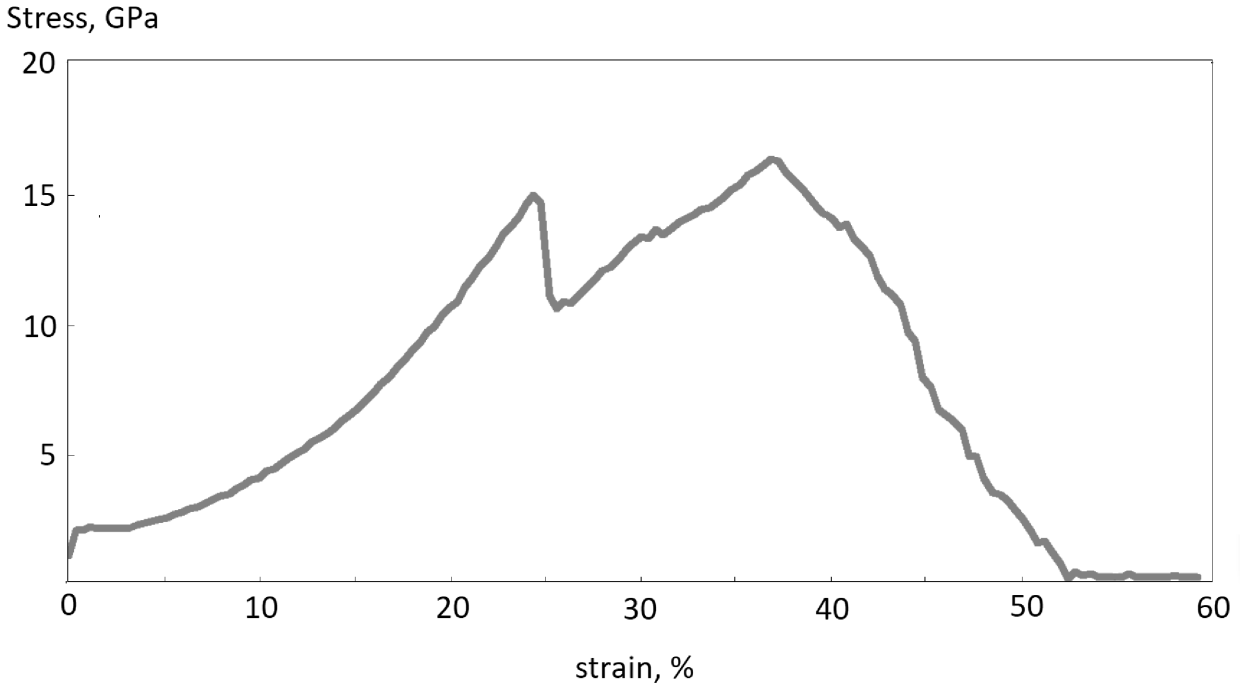
and then in the central part of the specimen. One can see that the hypothetical CNT fabric demonstrates the Yield strength of 15 GPa, which is three orders of magnitude higher that the strength of polymer films with similar flexibility and thickness [27]. Such exceptional properties are undoubtedly of a great interest for practical applications.

Conclusion

In this work we have presented the new general framework for modeling fibrous composites and textiles. Separate fibers are modeled as chains of interacting rigid bodies. The elasticity of individual fiber is represented with EVM formalism. The suggested framework is illustrated in the application to modeling of hypothetical CNT nanofabrics. In the benchmark example, the framework was capable to simulate relaxation and mechanical test on a CNT fabric specimen (2.4×10^6 model degrees of freedom, approximately 10^{12} contact resolution computations) in approximately 20 hours on 20 nodes, with nearly linear scaling with the number of cores used. In the benchmark example considered, HPC capabilities of our framework allowed to discover stabilization of large specimens of hypothetical CNT fabric by vdW adhesion forces, and to perform the mechanical test indicating superior properties of hypothetical CNT textile. The proposed framework is capable to efficiently model any systems of interacting elastic fibers at any length and time scales admitting athermal description. Any types of contact interactions



a) 0% strain b) 20% strain c) 40% strain



d) Stress-strain curve monitored during the test

Figure 4. Mechanical test on a CNT fabric specimen (a, b, c) – structure snapshots for engineering strains of 0%, 20%, and 40% respectively, d) – stress-strain curve

and nonlinearities in fiber's constitutive behaviours can be straightforwardly incorporated into suggested modeling concept. Therefore, our framework can be straightforwardly applied to a wide class of problems, including composites, ropes and textiles for aerospace and military applications.

Acknowledgements

Author acknowledges the financial support from the Russian Foundation for Basic Research (RFBR) under grants 16-31-60100 and 18-29-19198. Assistance of waLBerla package developers (S. Eibl and U. Rude) and Skoltech HPC team (R. Arslanov, I. Zacharov) is deeply appreciated. High-performance computations presented in the paper were carried out on Skoltech HPC cluster Zhores.

References

1. Iijima, S.: Helical microtubules of graphitic carbon. Nature 354, 56–58 (1991), DOI: 10.1038/354056a0

2. Baughman, R.: Carbon nanotubes – the route towards application. Science 297, 787–792 (2002), DOI: 10.1126/science.1060928

3. Eppell, S., Smith, B., Kahn, H., Ballarini, R.: Nano measurements with micro-devices: mechanical properties of hydrated collagen fibrils. Journ. Roy. Soc. Int. 3(6), 117–121 (2006), DOI: 10.1098/rsif.2005.0100

4. Chand, S.: Review carbon fibers for composites. Journ. Mater. Sci. 35(6), 1303–1313 (2000), DOI: 10.1023/a:1004780301489

5. Yakobson, B., Brabec, C., Bernholc, J.: Nanomechanics of Carbon Tubes: Instabilities beyond Linear Response. Phys. Rev. Lett. 76(14), 2511 (1996), DOI: 10.1103/physrevlett.76.2511

6. Dumitrica, T., Hua, M., Yakobson, B.: Symmetry-, time-, and temperature-dependent strength of carbon nanotubes. Proc. Natl. Acad. Sci. U.S.A. 103(16), 6105 (2006), DOI: 10.1073/pnas.0600945103

7. Zhang, D., Dumitrica, T.: Elasticity of ideal single-walled carbon nanotubes via symmetry-adapted tight-binding objective modeling. Appl. Phys. Lett., 93, 031919 (2008), DOI: 10.1063/1.2965465

8. Nikiforov, I., Zhang, D., James, R., Dumitrica, T.: Wavelike rippling in multiwalled carbon nanotubes under pure bending. Appl. Phys. Lett. 96, 123107 (2010), DOI: 10.1063/1.3368703

9. Buehler, M.: Mesoscale modeling of mechanics of carbon nanotubes: Self-assembly, self-folding and fracture. Journ. Mat. Res. 21(11), 2855 (2006), DOI: 10.1557/jmr.2006.0347

10. Cranford, S., Buehler, M.: In silico assembly and nanomechanical characterization of carbon nanotube buckypaper. Nanotechnology 21, 265706 (2010), DOI: 10.1088/0957-4484/21/26/265706

11. Mirzaeifar, R., Qin, Z., Buehler, M.: Mesoscale mechanics of twisting carbon nanotube yarns. Nanoscale 7(12), 5435 (2015), DOI: 10.1039/c4nr06669c

12. Anderson, T., Akatyeva, E., Nikiforov, I., Potyondy, D., Ballarini, R., Dumitrica, T.: Toward distinct element method simulations of carbon nanotube systems. Journ. Nanotech. Eng. Med. 1(4), 0410009 (2010), DOI: 10.1115/1.4002609

13. Ostanin, I., Ballarini, R., Potyondy, D., Dumitrica, T.: A distinct element method for large scale simulations of carbon nanotube assemblies. Mech. Phys. Sol. 61(3), 762 (2013), DOI: 10.1016/j.jmps.2012.10.016

14. Ostanin, I., Ballarini, R., Dumitrica, T.: Distinct element method modeling of carbon nanotube bundles with intertube sliding and dissipation. Appl. Mech. 81(6), 061004 (2014), DOI: 10.1115/1.4026484

15. Wang, Y., Gaidau, C., Ostanin, I., Dumitrica, T.: Ring windings from single-wall carbon nanotubes: A distinct element method study. Appl. Phys. Lett. 103 (18), 183902 (2013), DOI: 10.1063/1.4827337

16. Wang, Y., Semler, M., Ostanin, I., Hobbie, E., Dumitrica, T.: Rings and rackets from single-wall carbon nanotubes: manifestations of mesoscale mechanics. Soft Matter 10 (43), 8635–8640 (2014), DOI: 10.1038/354056a0

17. Ostanin, I., Ballarini, R., Dumitrica, T.: Rings and rackets from single-wall carbon nanotubes: manifestations of mesoscale mechanics. Journ. Mat. Res. 30(1), 19 (2015), DOI: 10.1039/c4sm00865k

18. Wang, Y., Ostanin, I., Gaidau, C., Dumitrica, T.: Twisting carbon nanotube ropes with the mesoscopic distinct element method: Geometry, packing, and nanomechanics. Langmuir, 31(45), 12323 (2015), DOI: 10.1021/acs.langmuir.5b03208

19. Ostanin, I., Zhilyaev, P., Petrov, V., Dumitrica, T., Eibl, S., Ruede, U., Kuzkin, V.: Toward large scale modeling of carbon nanotube systems with the mesoscopic distinct element method. Mater. 8(3), 240–245 (2018), DOI: 10.22226/2410-3535-2018-3-240-245

20. Preclik, T., Ruede U.: Ultrascale simulations of non-smooth granular dynamics. Comp. Part. Mech., 2, 173 (2015), DOI: 10.1007/s40571-015-0047-6

21. Itasca Consulting Group Inc., 2015. PFC3D (Particle Flow Code in Three Dimensions). Version 5.0. Itasca Consulting Group Inc., Minneapolis

22. Kuzkin, V., Asonov, I.: Vector-based model of elastic bonds for simulation of granular solids. Phys. Rev. E, 86(5), 051301 (2012), DOI: 10.1103/physreve.86.051301

23. Kuzkin, V., Krivtsov, A.: Enhanced vector-based model for elastic bonds in solids. Lett. Mat. 7(4), 455 (2017), DOI: 10.22226/2410-3535-2017-4-455-458

24. MPI Forum. MPI: A message-passing interface standard. Technical report, Knoxville, TN, USA (1994)

25. Ericson, C.: Real-time collision detection. CRC Press (2004)

26. Zacharov, I., Arslanov, R., Gunin, M., Stefonshin, D., Pavlov, S., Panarin, O., Maliutin, A., Rykovanov., S.: "Zhores" — new PFlops supercomputer for data-driven modeling, machine learning and artificial intelligence installed in Skolkovo Institute of Science and Technology. Preprint arxiv 1902.07490 (2018)

27. Huang, C.K., Lou, W.M., Tsai, C.J., Wu, T.C., Lin, H.Y.: Mechanical properties of polymer thin film measured by the bulge test. Thin Sol. Films 515, 7222 (2007), DOI: 10.1016/j.tsf.2007.01.058